

FIG. 1

$$\square rd_{128} = m[rc](128*64/size) * rb_{128}$$

$$m[rc](128*64/size)$$

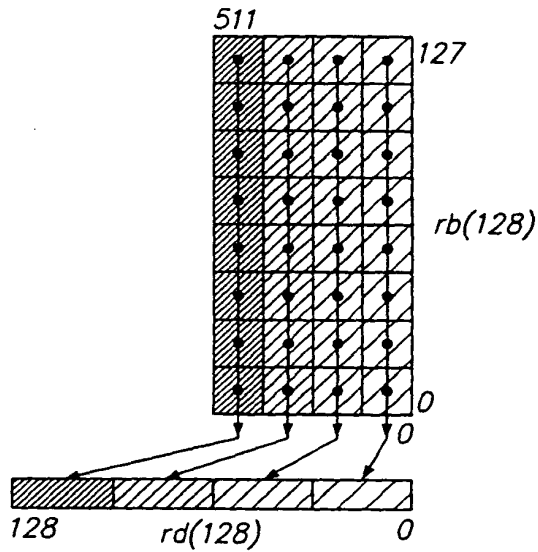


FIG. 2

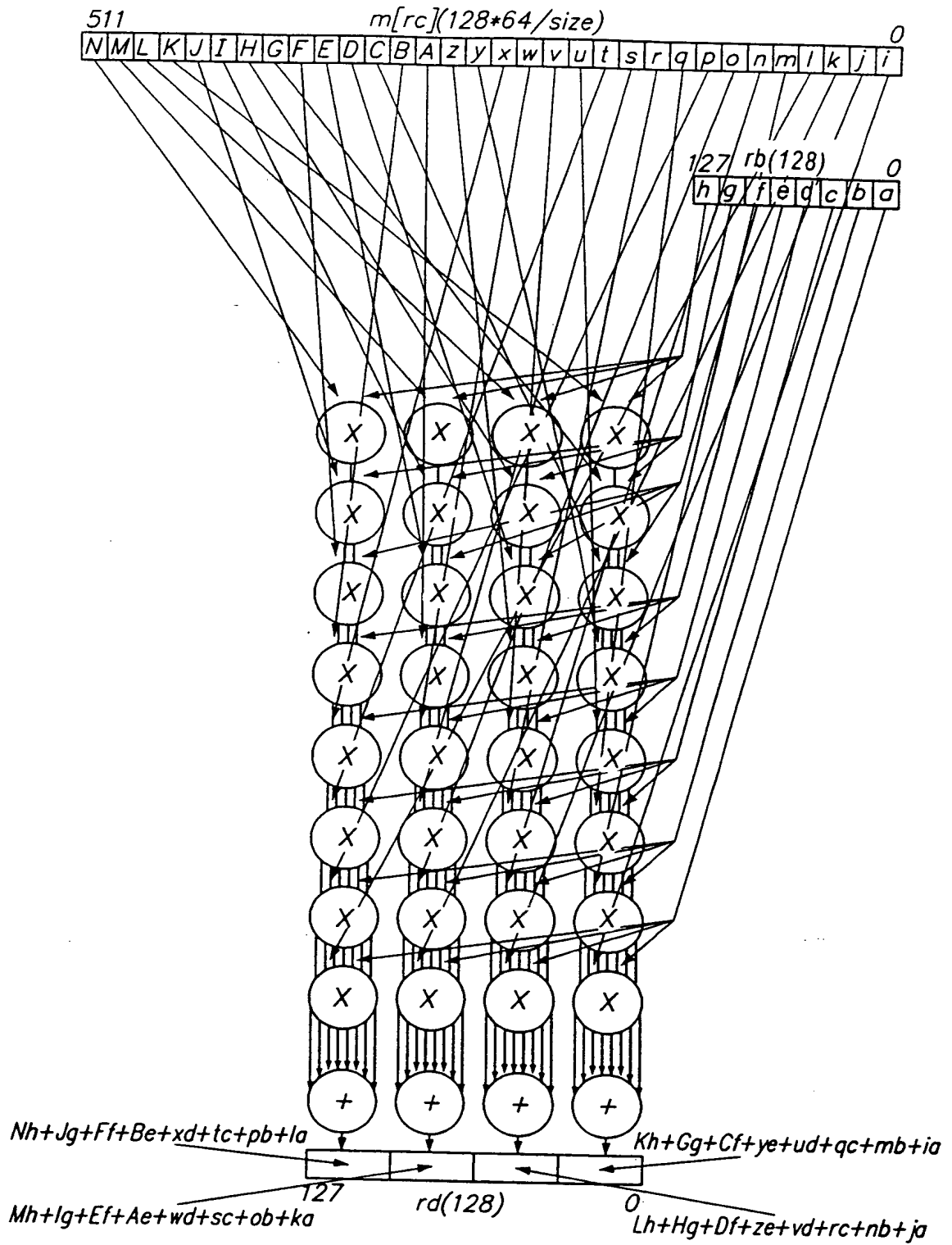


FIG. 3

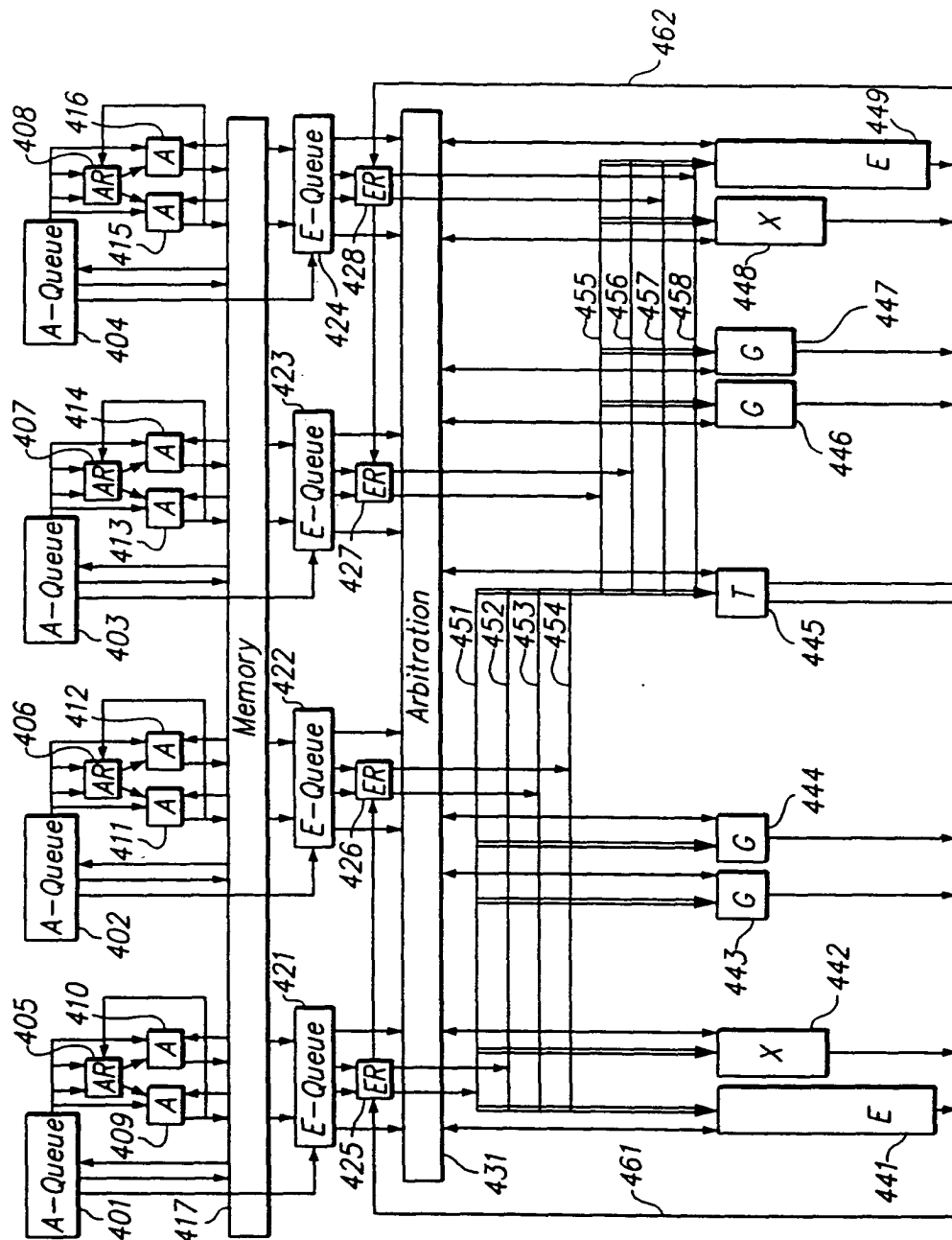
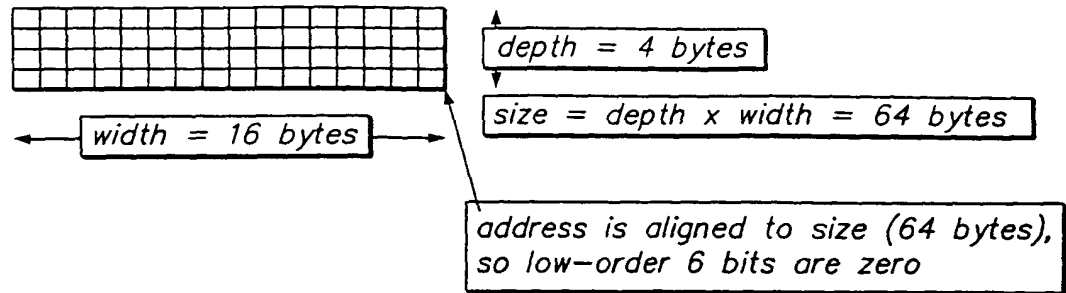


FIG. 4

$$\square \text{ specifier} = \text{address} + (\text{size}/2) + (\text{width}/2)$$



address	aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	000000
size/2	00000000000000000000000000000000	100000
width/2	00000000000000000000000000000000	001000

specifier	aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	101000
-----------	----------------------------------	--------

500 505 510

FIG. 5

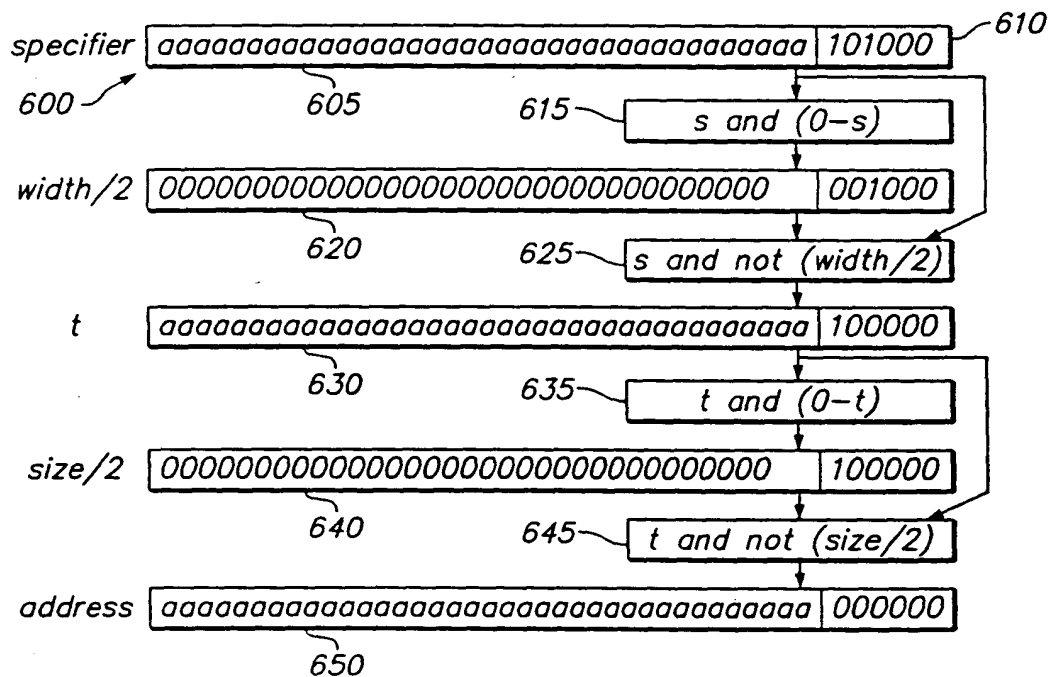


FIG. 6

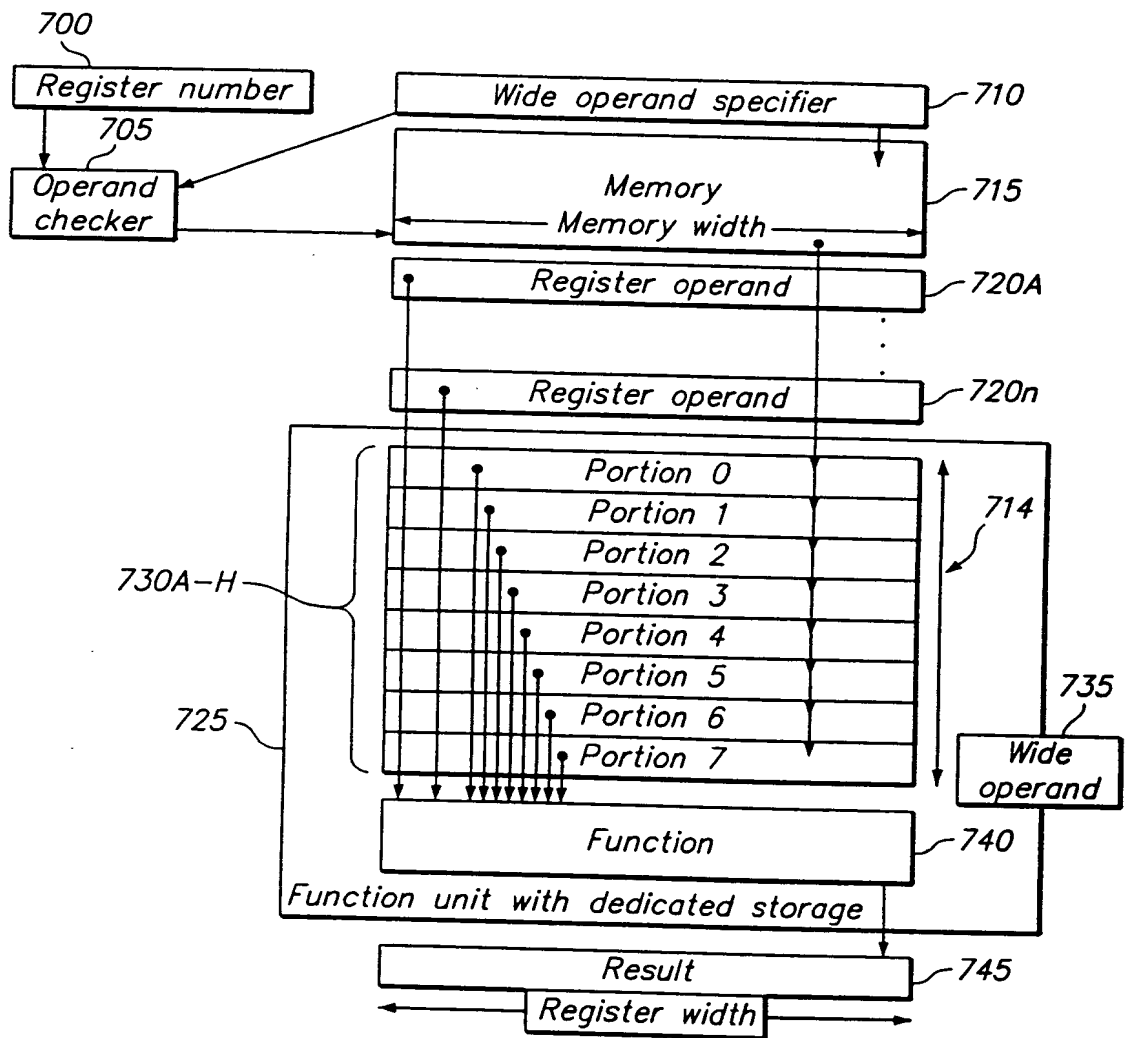


FIG. 7

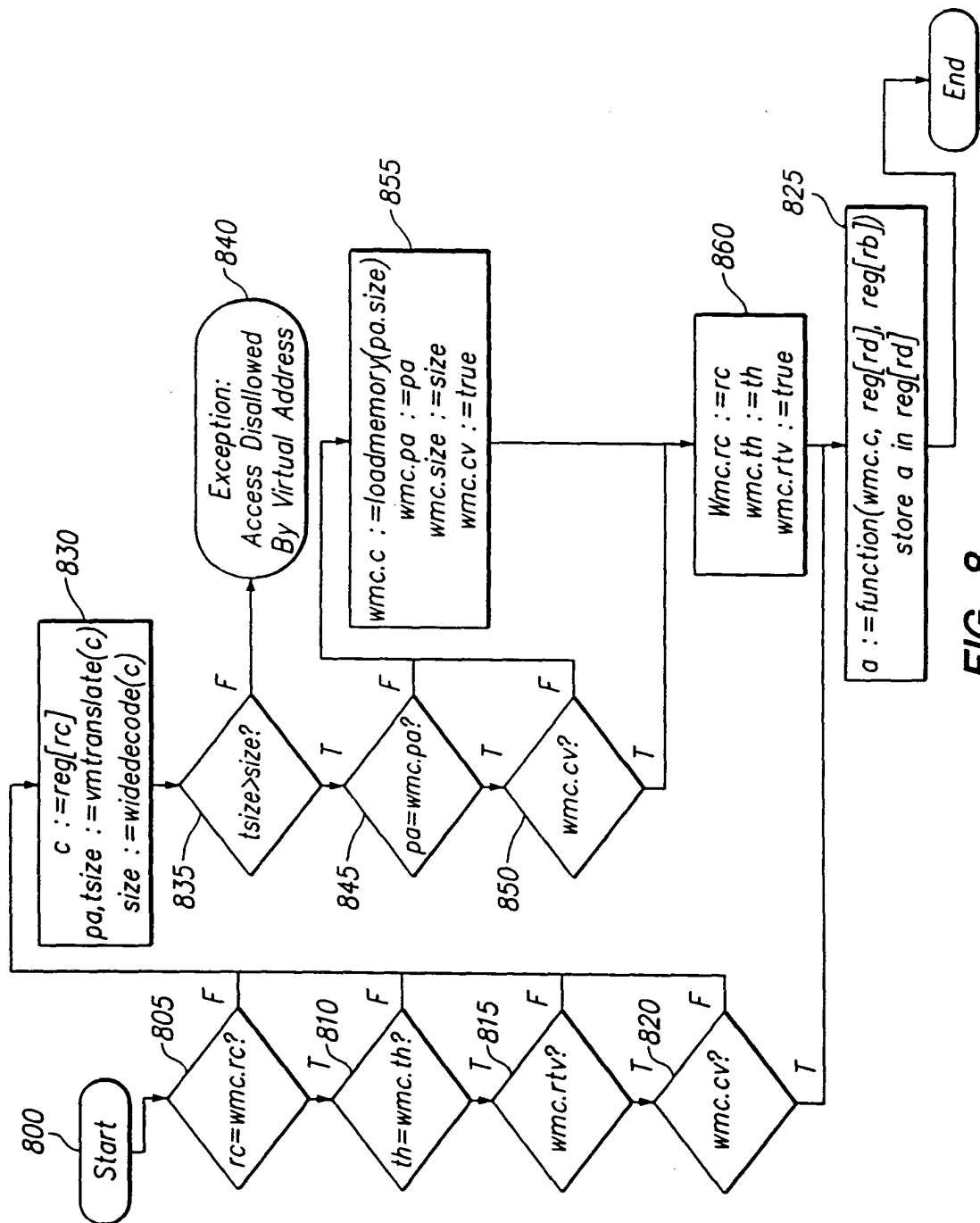
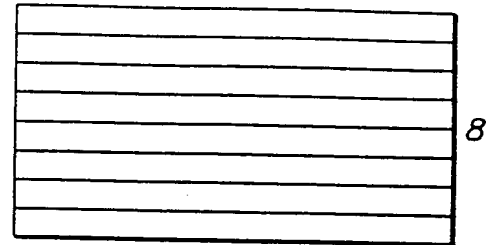


FIG. 8

☐ *wmc.c contents*



128

☐ *wmc.pa—physical address*



64

☐ *wmc.size—size of contents*



10

☐ *wmc.cv—contents valid*



1

☐ *wmc.th—thread last used*



2

☐ *wmc.reg—register last used*



6

☐ *wmc.rtv—register & thread valid*



1

FIG. 9

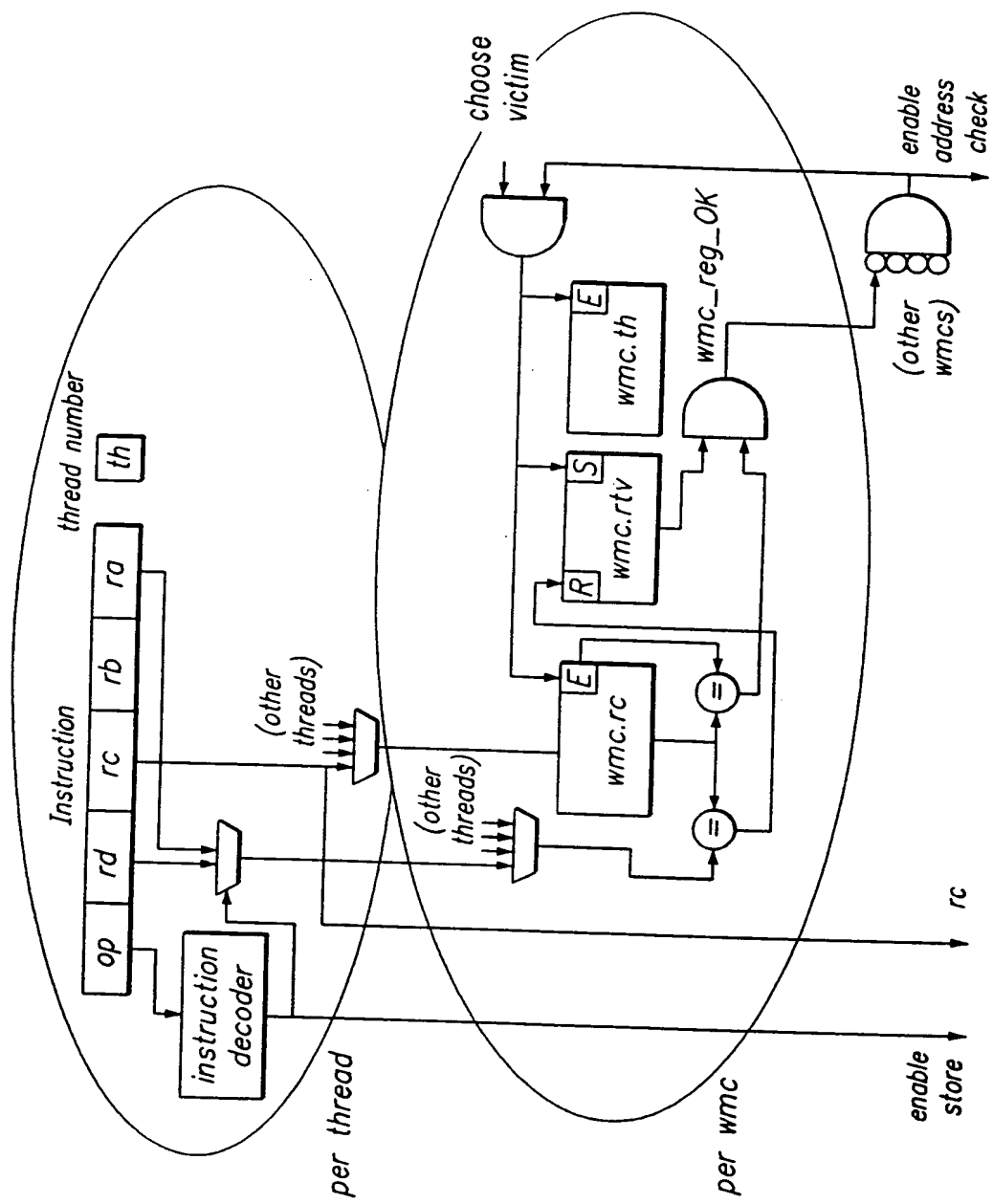
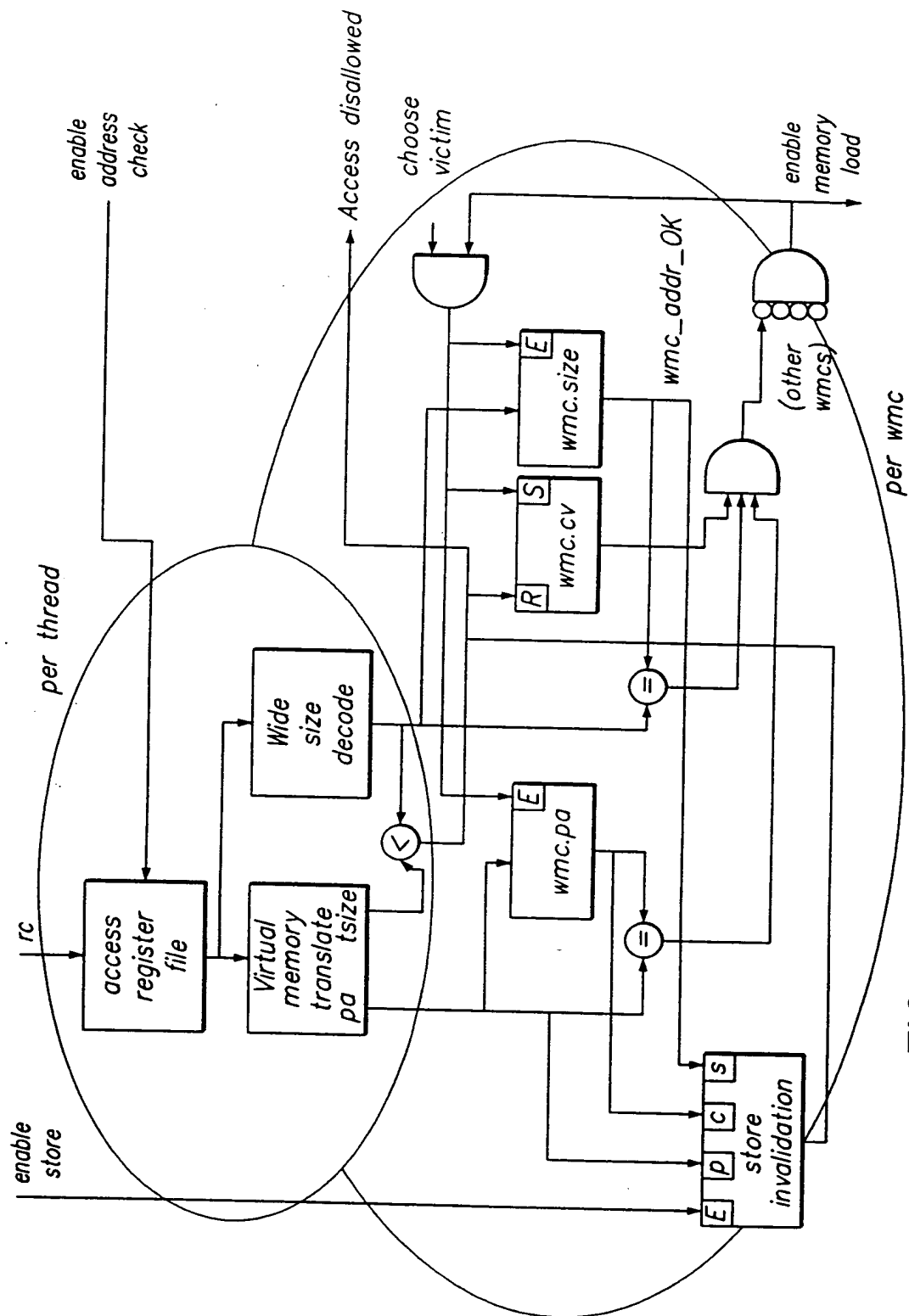


FIG. 10



210

Operation codes

W.SWITCH.B	Wide switch big-endian
W.SWITCH.L	Wide switch little-endian

Selection

class	op	order
Wide switch	W.SWITCH	B L

Format

W.op.order ra=rc,rd,rb

ra=woporder(rc,rd,rb)

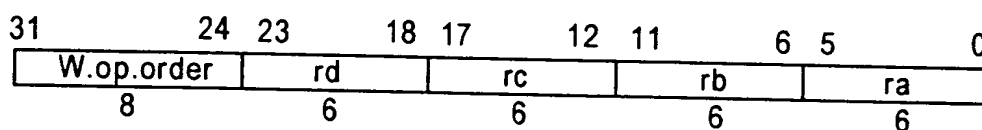


FIG. 12A

1230

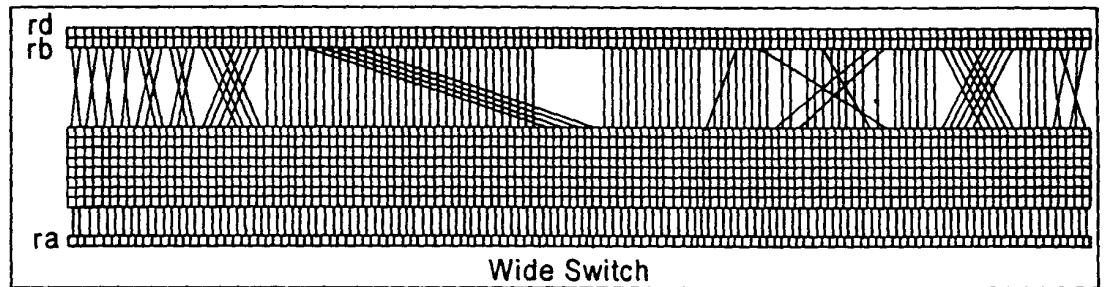


FIG. 12B

Definition

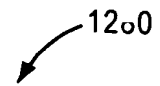
```

defWideSwitch(op,rd,rc,rb,ra)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 128)
  if c1..0 ≠ 0 then
    raise AccessDisallowedByVirtual Address
  elseif c6..0 ≠ 0 then
    VirtAddr ← c and (c-1)
    W ← wsize ← (c and (0-c)) || 01
  else
    VirAddr ← c
    w ← wsize ← 128
  endif
  msize ← 8*wsize
  lwsiz ← log(wsize)
  case op of
    W.SWITCH.B:
      order ← B
    W.SWITCH.L:
      order ← L
  endcase
  m ← LoadMemory(c, VirtAddr,msize,order)
  db ← d || b
  for i ← 0 to 127
    j ← 0 || i1wsiz-1..0
    k ← m7*w+j || m6*w+j || m5*w+j || m4*w+j || m3*w+j || m2*w+j || mw+j || mj
    l ← i7..1wsiz || j1wsiz-1..0
    ai ← dbl
  endfor
  RegWrite(ra, 128, a)
enddef

```

FIG. 12C

1200



Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 12D

1210

Operation codes

W.TRANSLATE.8.B	Wide translate bytes big-endian
W.TRANSLATE.16.B	Wide translate doublets bit-endian
W.TRANSLATE.32.B	Wide translate quadlets bit-endian
W.TRANSLATE.64.B	Wide translate octlets big-endian
W.TRANSLATE.8.L	Wide translate bytes little-endian
W.TRANSLATE.16.L	Wide translate doublets little-endian
W.TRANSLATE.32.L	Wide translate quadlets little-endian
W.TRANSLATE.64.L	Wide translate octlets little-endian

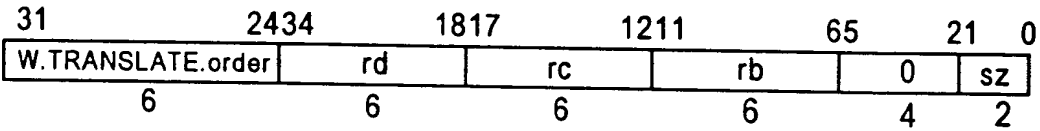
Selection

class	size	order
Wide translate	8 16 32 64	B L

Format

W.TRANSLATE.size.order rd=rc,rb

rd=wtranslatesizeorder(rc,rb)



sz ← log(size) = 3

FIG. 13A

1330

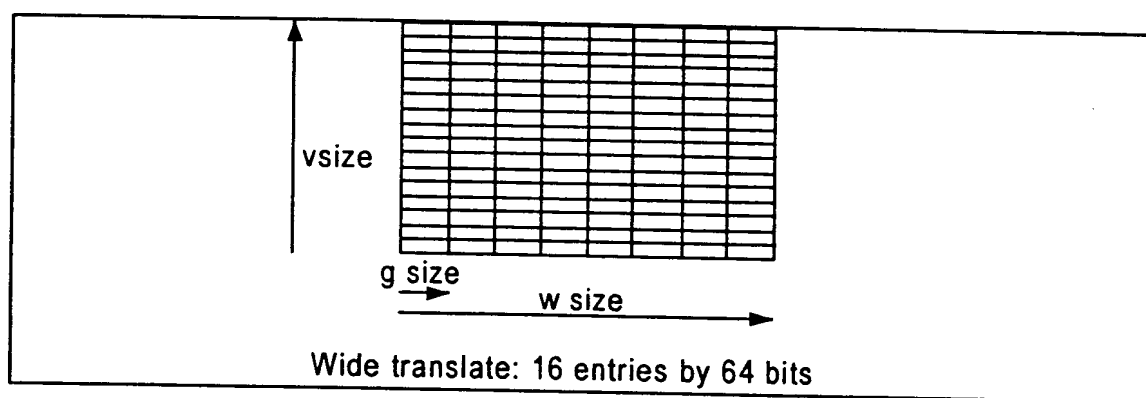


FIG. 13B

Definition

```

def Wide Translate(op, gsize, rd, rc, rb)
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 128)
  lsize ← log(gsize)
  if clsize-4..0 ≠ 0 then
    raise AccessDisallowedByVirtual Address
  endif
  if c4..lsize-3 ≠ 0 then
    wsize ← (c and (0-c)) || 03
    t ← c and (c-1)
  else
    wsize ← 128
    t ← c
  endif
  lsize ← log(wsize)
  if tlsize+4..lsize-2 ≠ 0 then
    msize ← (t and (0-t)) || 04
    VirtAddr ← t and (t-1)
  else
    msize ← 256*wsize
    VirtAddr ← t
  endif
  case op of
    W.TRANSLATE.B:
      order ← B
    W.TRANSLATE.L:
      order ← L
  endcase
  m ← LoadMemory(c, VirtAddr, msize, order)
  vsize ← msize/wsize
  lvsize ← log(vsize)
  for i ← 0 to 128-gsize by gsize
    j ← ((order=B)lvsize )^(blvsize-1+i..i ) * wsize + ilvsize-1..0
    agsize-1+i..i ← mj+gsize-1..j
  endfor
  RegWrite(rd, 128, a)
enddef

```

FIG. 13C

1380



Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 13D

Operation codes

1410

W.MUL.MAT.8.B	Wide multiply matrix signed byte big-endian
W.MUL.MAT.8.L	Wide multiply matrix signed byte little-endian
W.MUL.MAT.16.B	Wide multiply matrix signed doublet big-endian
W.MUL.MAT.16.L	Wide multiply matrix signed doublet little-endian
W.MUL.MAT.32.B	Wide multiply matrix signed quadlet big-endian
W.MUL.MAT.32.L	Wide multiply matrix signed quadlet little-endian
W.MUL.MAT.C.8.B	Wide multiply matrix signed complex byte big-endian
W.MUL.MAT.C.8.L	Wide multiply matrix signed complex byte little-endian
W.MUL.MAT.C.16.B	Wide multiply matrix signed complex doublet big-endian
W.MUL.MAT.C.16.L	Wide multiply matrix signed complex doublet little-endian
W.MUL.MAT.M.8.B	Wide multiply matrix mixed-signed byte big-endian
W.MUL.MAT.M.8.L	Wide multiply matrix mixed-signed byte little-endian
W.MUL.MAT.M.16.B	Wide multiply matrix mixed-signed doublet big-endian
W.MUL.MAT.M.16.L	Wide multiply matrix mixed-signed doublet little-endian
W.MUL.MAT.M.32.B	Wide multiply matrix mixed-signed quadlet big-endian
W.MUL.MAT.M.32.L	Wide multiply matrix mixed-signed quadlet little-endian
W.MUL.MAT.P.8.B	Wide multiply matrix polynomial byte big-endian
W.MUL.MAT.P.8.L	Wide multiply matrix polynomial byte little-endian
W.MUL.MAT.P.16.B	Wide multiply matrix polynomial doublet big-endian
W.MUL.MAT.P.16.L	Wide multiply matrix polynomial doublet little-endian
W.MUL.MAT.P.32.B	Wide multiply matrix polynomial quadlet big-endian
W.MUL.MAT.P.32.L	Wide multiply matrix polynomial quadlet little-endian
W.MUL.MAT.U.8.B	Wide multiply matrix unsigned byte big-endian
W.MUL.MAT.U.8.L	Wide multiply matrix unsigned byte little-endian
W.MUL.MAT.U.16.B	Wide multiply matrix unsigned doublet big-endian
W.MUL.MAT.U.16.L	Wide multiply matrix unsigned doublet little-endian
W.MUL.MAT.U.32.B	Wide multiply matrix unsigned quadlet big-endian
W.MUL.MAT.U.32.L	Wide multiply matrix unsigned quadlet little-endian

Selection

class	op	type	size	order
multiply	W.MUL.MAT	NONE MUP	8 16 32	B
				L
		C	8 16	B
				L

Format

W.op.size.order rd=rc,rb

rd=wopsizeorder(rc,rb)

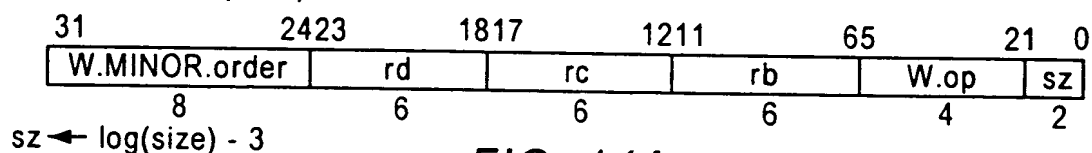


FIG. 14A

1430

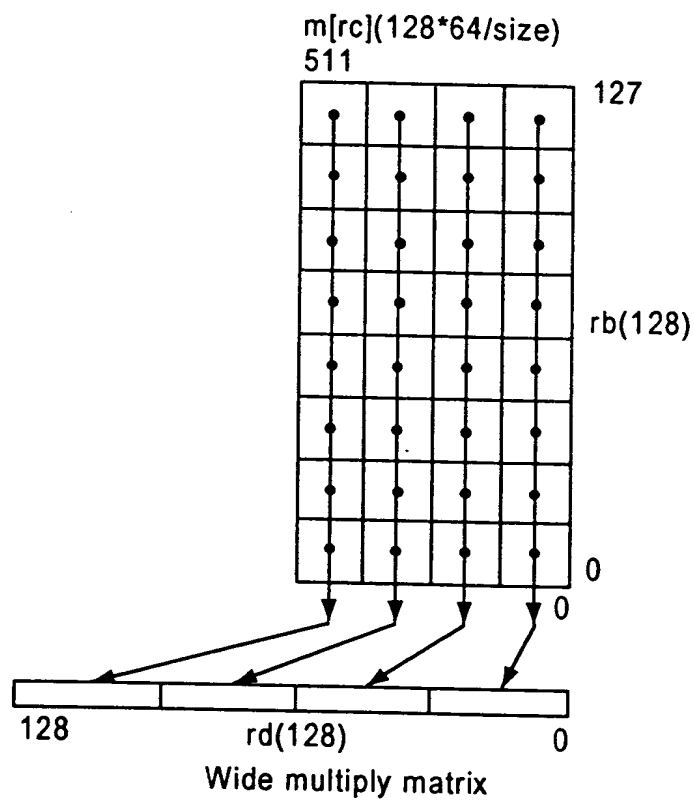


FIG. 14B

1460

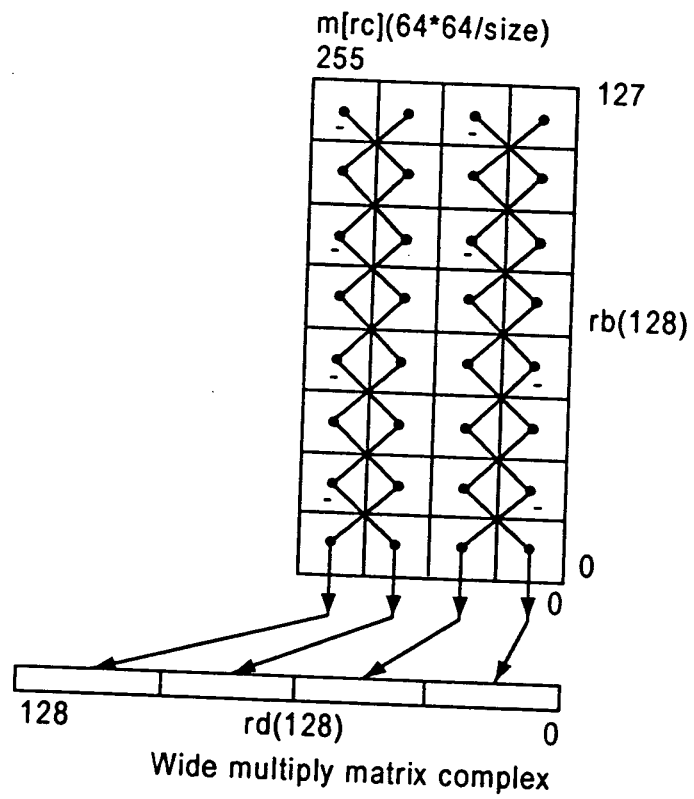


FIG. 14C

Definition

```

def mul(size,h,vs,v,i,ws,j)as
  mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsize-1+j)h-size || wsize-1+j..j)
enddef

def c ← PolyMultiply(size,a,b) as
  p[0] ← 02*size
  for k ← 0 to size-1
    p[k+1] ← p[k] ^ ak ? (0size-k || b || 0k) : 02*size
  endfor
  c ← p[size]
enddef

def WideMultiplyMatrix(major,op,gsize,rd,rc,rb)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 64)
  b ← RegRead(rb,128)
  lsize ← log(gsize)
  if clsize-4..0 ≠ 0 then
    raise AccessDisallowedByVirtualAddress
  endif
  if c2..lsize-3 ≠ 0 then
    wsize ← (c and (0-c)) || 04
    t ← c and (c-1)
  else
    wsize ← 64
    t ← a
  endif
  lsize ← log(wsize)
  if tlsize+6-lsize..lsize-3 ≠ 0 then
    msize ← (t and (0-t)) || 04
    VirtAddr ← t and (t-1)
  else
    msize ← 128*wsize/gsize
    VirtAddr ← t
  endif
  case major of
    W.MINOR.B:
      order ← B
    W.MINOR.L:
      order ← L
  endcase
endcase

```

FIG. 14D-1

```

case op of
  M.MUL.MAT.U.8, W.MUL.MAT.U.16, W.MUL.MAT.U.32,
  W.MUL.MAT.U.64:
    ms ← bs ← 0
    W.MUL.MAT.M.8, W.MUL.MAT.M.16, W.MUL.MAT.M.32,
    W.MUL.MAT.M.64
    ms ← 0
    bs ← 1
    W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32,
    W.MUL.MAT.64, W.MUL.MAT.C.8, W.MUL.MAT.C.16,
    W.MUL.MAT.C.32, W.MUL.MAT.C.64:
    ms ← bs ← 1
    W.MUL.MAT.P.8, W.MUL.MAT.P.16, W.MUL.MAT.P.32,
    W.MUL.MAT.P.64:
endcase
m ← LoadMemory(c,VirtAddr,msize,order)
h ← 2*gszsize

for i ← 0 to wsize-gsize by gsize
  q[0] ← 02*gszsize
  for j ← 0 to vsize-gsize by gsize
    case op of
      W.MUL.MAT.P.8, W.MUL.MAT.P.16,
      W.MUL.MAT.P.32, W.MUL.MAT.P.64:
        k ← i+wsize*j8..lgsize
        q[j+gszsize] ← q[j] ^ PolyMultiply(gsize,mk+gszsize-1..k,
        bj+gszsize-1..j)
      W.MUL.MAT.C.8, W.MUL.MAT.C.16, W.MUL.MAT.C.32,
      W.MUL.MAT.C.64:
        if (~i) & gsize = 0 then
          k ← i-(j&gszsize)+wsize*j8..lgsize+1
          q[j+gszsize] ← q[i] + mul(gsize,h,ms,m,k,bs,b,j)
        else
          k ← i+gszsize+wsize*j8..lgsize+1
          q[i+gszsize] ← q[i] = mul(gsize,h,ms,m,k,bs,b,j)
        endif
  endfor
endfor


```

FIG. 14D-2

W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32,
W.MUL.MAT.64, W.MUL.MAT.M.8, W.MUL.MAT.M.16,
W.MUL.MAT.M.32, W.MUL.MAT.M.64, W.MUL.MAT.U.8,
W.MUL.MAT.U.16, W.MUL.MAT.U.32, W.MUL.MAT.U.64
q[i+gsize] ← q[i] + mul(gsize,h,ms,m,i+wsiz*
j8..lgsiz,bs,b,j)
endfor
a2*gsiz-1+2*i..2*i ← q[vsize]
endfor
a127..2*wsiz ← 0
RegWrite(rd, 128, a)
enddef

FIG. 14D-3

1490



Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 14E

1510 ↙

Operation codes

W.MUL.MAT.X.B	Wide multiply matrix extract big-endian
W.MUL.MAT.X.L	Wide multiply matrix extract little-indian

Selection

class	op	order
Multiply matrix extract	W.MUL.MAT.X	B L

Format

W.op.order ra=rc,rd,rb

ra=wop(rc,rd,rb)

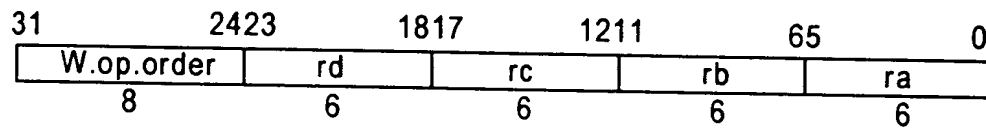


FIG. 15A

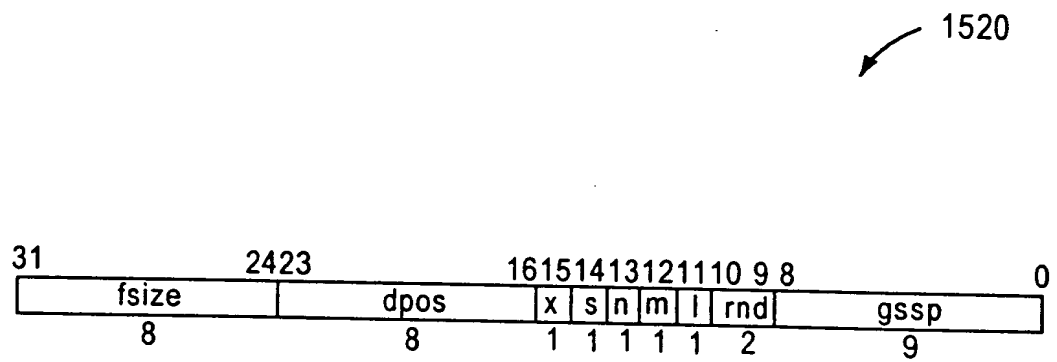


FIG. 15B

1530

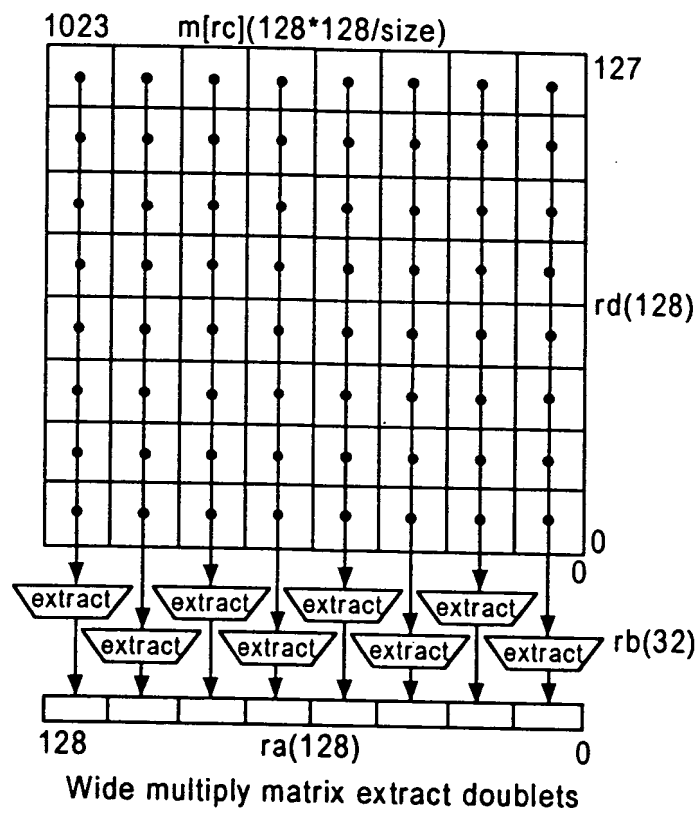


FIG. 15C

1560

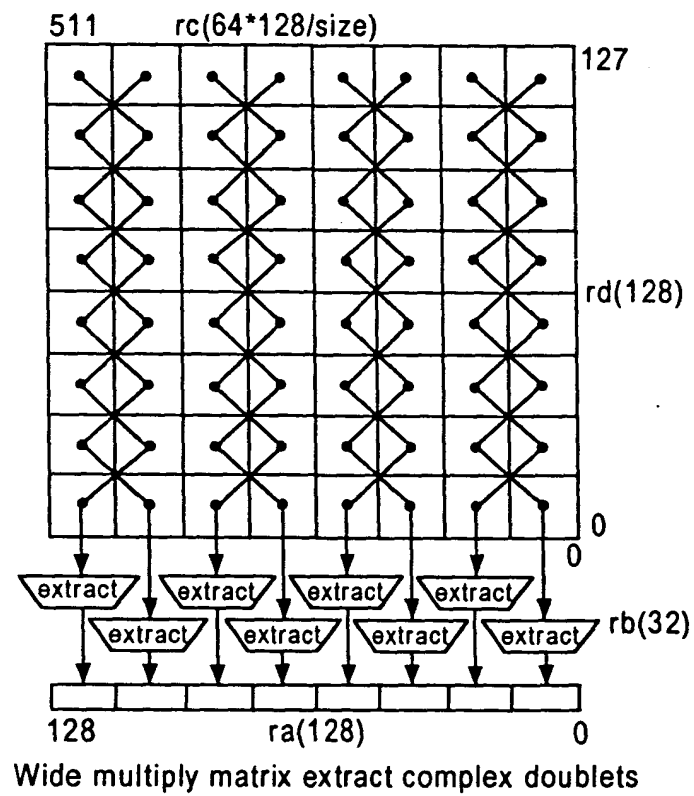


FIG. 15D

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as  
    mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsize-1+j)h-size || wsize-1+j..j)  
enddef
```

1580

```
def WideMultiplyMatrixExtract(op,ra,rb,rc,rd)
```

```
    d ← RegRead(rd, 128)
```

```
    c ← RegRead(rc, 64)
```

```
    b ← RegRead(rb, 128)
```

```
    case b8..0 of
```

```
        0..255:
```

```
            sgsz ← 128
```

```
        256..383:
```

```
            sgsz ← 64
```

```
        384..447:
```

```
            sgsz ← 32
```

```
        448..479:
```

```
            sgsz ← 16
```

```
        480..495:
```

```
            sgsz ← 8
```

```
        496..503:
```

```
            sgsz ← 4
```

```
        504..507:
```

```
            sgsz ← 2
```

```
        508..511:
```

```
            sgsz ← 1
```

```
    endcase
```

```
    l ← b11
```

```
    m ← b12
```

```
    n ← b13
```

```
    signed ← b14
```

```
    if c3..0 ≠ 0 then
```

```
        wsize ← (c and (0-c)) || 04
```

```
        t ← c and (c-1)
```

```
    else
```

```
        wsize ← 128
```

```
        t ← c
```

```
    endif
```

```
    if sgsz < 8 then
```

```
        gsize ← 8
```

```
    elseif sgsz > wsize/2 then
```

```
        gsize ← wsize/2
```

```
    else
```

FIG. 15E-1

```

        gsize ← sysize
    endif
    lysize ← log(gsize)
    lwsiz ← log(wsize)
    if tlwsiz+6-n-lysize..lwsiz-3 ≠ 0 then
        msize ← (t and (0-t)) || 04
        VirtAddr ← t and (t-1)
    else
        msize ← 64*(2-n)*wsiz/gsize
        VirtAddr ← t
    endif
    vsiz ← (1+n)*msiz*gsiz/wsiz
    mm ← LoadMemory(c,VirtAddr,msiz,order)
    lmsiz ← log(msiz)
    if (VirtAddrlmsiz-4..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    case op of
        W.MUL.MAT.X.B:
            order ← B
        W.MUL.MAT.X.L:
            order ← L
    endcase
    ms ← signed
    ds ← signed ^ m
    as ← signed or m
    spos ← (b8..0) and (2*gsiz-1)
    dpos ← (0 || b23..16) and (gsiz-1)
    r ← spos
    sfsiz ← (0 || b31..24) and (gsiz-1)
    tfsiz ← (sfsiz = 0) or ((sfsiz+dpos) > gsiz) ? gsiz-dpos : sfsiz
    fsiz ← (tfsiz + spos > h) ? h - spos : tfsiz
    if (b10..9 = Z) & ~signed then
        rnd ← F
    else
        rnd ← b10..9
    endif

```

FIG. 15E-2

1580

```

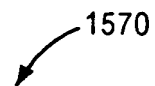
for i ← 0 to wsize-gsize by gsize
  q[0] ← 02*gsize+7-lgsize
  for j ← 0 to vsize-gsize by gsize
    if n then
      if (~) & j & gsize = 0 then
        k ← i-(j&gsize)+wsize*j8..lgsize+1
        q[i+gsize] ← q[i] + mul(gsize,h,ms,mm,k,ds,d,j)
      else
        k ← i+gsize+wsize*j8..lgsize+1
        q[i+gsize] ← q[i] - mul(gsize,h,ms,mm,k,ds,d,j)
      endif
    else
      q[i+gsize] ← q[i] = mul(gsize,h,ms,mm,i+j*wsize/gsize,ds,d,j)
    endif
  endfor
  p ← q[128]
  case rnd of
    none, N:
      s ← 0h-r || ~pr || prr-1
    Z:
      s ← 0h-r || ph-1r
    F:
      s ← 0h
    C:
      s ← 0h-r || 1r
  endcase
  v ← ((ds & ph-1) || p) + (0 || s)

  if (vh..r+fsize = (as & vr+fsize-1)h+1-r-fsize) or not I then
    w ← (as & vr+fsize-1)gsize-fsize-dpos || vfsize-1+r..r || 0dpos
  else
    w ← (s ? (vh || ~vhgsize-dpos-1) : 1gsize-dpos) || 0dpos
  endif
  asize-1+i..i ← w
endfor
a127..wsize ← 0
RegWrite(ra, 128, a)
enddef

```

FIG. 15E-3

1570



Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 15F

Operation codes

W.MUL.MAT.X.I.8.B	Wide multiply matrix extract immediate signed byte big-endian
W.MUL.MAT.X.I.8.L	Wide multiply matrix extract immediate signed byte little-endian
W.MUL.MAT.X.I.16.B	Wide multiply matrix extract immediate signed doublet big-endian
W.MUL.MAT.X.I.16.L	Wide multiply matrix extract immediate signed doublet little-endian
W.MUL.MAT.X.I.32.B	Wide multiply matrix extract immediate signed quadlet big-endian
W.MUL.MAT.X.I.32.L	Wide multiply matrix extract immediate signed quadlet little-endian
W.MUL.MAT.X.I.64.B	Wide multiply matrix extract immediate signed octlets big-endian
W.MUL.MAT.X.I.64.L	Wide multiply matrix extract immediate signed octlets little-endian
W.MUL.MAT.X.I.C.8.B	Wide multiply matrix extract immediate complex bytes big-endian
W.MUL.MAT.X.I.C.8.L	Wide multiply matrix extract immediate complex bytes little-endian
W.MUL.MAT.X.I.C.16.B	Wide multiply matrix extract immediate complex doublets big-endian
W.MUL.MAT.X.I.C.16.L	Wide multiply matrix extract immediate complex doublets little-endian
W.MUL.MAT.X.I.C.32.B	Wide multiply matrix extract immediate complex quadlets big-endian
W.MUL.MAT.X.I.C.32.L	Wide multiply matrix extract immediate complex quadlets little-endian

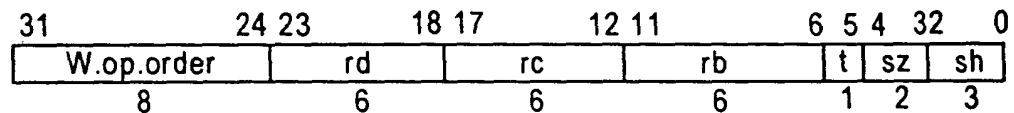
Selection

class	op	type	size	order
wide multiply extract immediate	W.MUL.MAT.X.I	NONE	8 16 32 64	L B
		C	8 16 32	L B

Format

W.op.tsize.order rd=rc,rb, i

rd=wopsizeorder(rc,rb,i)



sz ← log(size) - 3

assert size+3 ≥ i ≥ size-4

sh ← i - size

FIG. 16A

1630

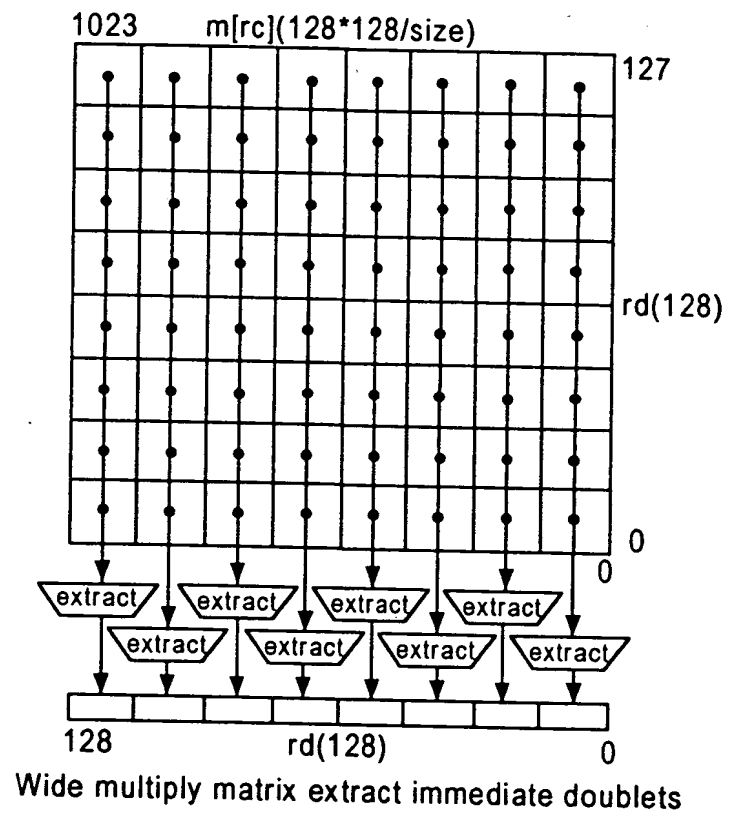


FIG. 16B

1660

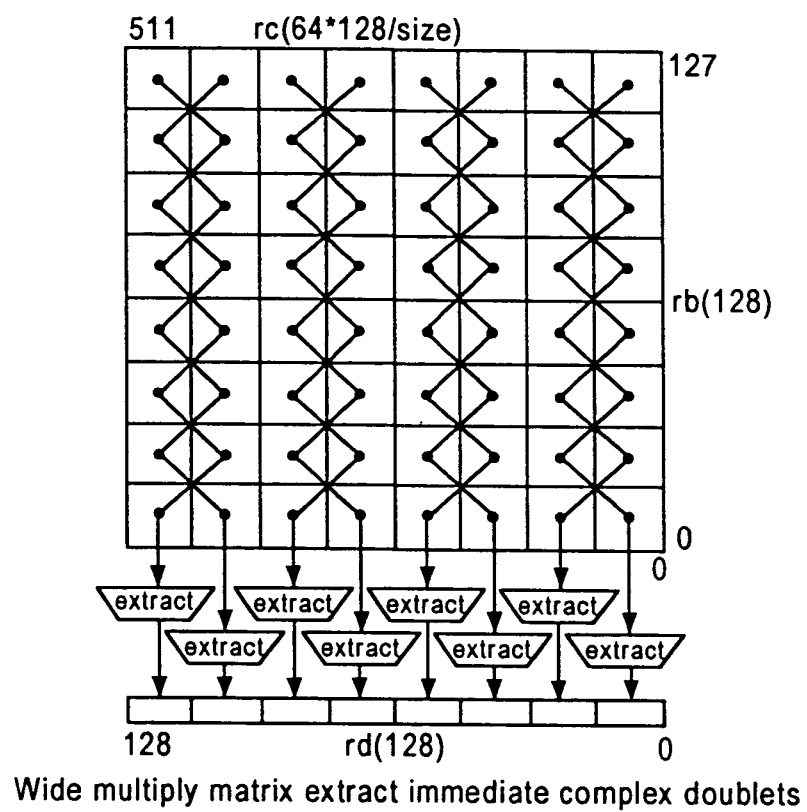


FIG. 16C

Definition

```

def mul(size,h,vs,v,i,ws,w,j) as
    mul  $\leftarrow ((vs \& v_{size-1+i})^{h-size} || v_{size-1+i..i}) * ((ws \& w_{size-1+j})^{h-size} || w_{size-1+j..j})$ 
enddef

```

```

def WideMultiplyMatrixExtractimmediate(op,type,gsize,rd,rc,rb,sh)

```

```

    c  $\leftarrow$  RegRead(rc, 64)
    b  $\leftarrow$  RegRead(rb, 128)
    lsize  $\leftarrow$  log(gsize)
    case type of
        NONE:
            if  $c_{lsize-4..0} \neq 0$  then
                raise AccessDisallowedBy VirtualAddress
            endif
            if  $c_{3..lsize-3} \neq 0$  then
                wsize  $\leftarrow (c \text{ and } (0-c)) || 0^4$ 
                t  $\leftarrow c \text{ and } (c-1)$ 
            else
                wsize  $\leftarrow$  128
                t  $\leftarrow$  c
            endif
            lwsz  $\leftarrow$  log(wsize)
            if  $t_{lwsz+6-lsize..lwsz-3} \neq 0$  then
                msize  $\leftarrow (t \text{ and } (0-t)) || 0^4$ 
                VirtAddr  $\leftarrow t \text{ and } (t-1)$ 
            else
                msize  $\leftarrow$  128*wsize/gsize
                VirtAddr  $\leftarrow$  t
        C:
            if  $c_{lsize-4..0} \neq 0$  then
                raise AccessDisallowedByVirtualAddress
            endif
            if  $c_{3..lsize-3} \neq 0$  then
                wsize  $\leftarrow (c \text{ and } (0-c)) || 0^4$ 
                t  $\leftarrow c \text{ and } (c-1)$ 
            else
                wsize  $\leftarrow$  128
                t  $\leftarrow$  c
            endif
            lwsz  $\leftarrow$  log(wsize)
            if  $t_{lwsz+5-lsize..lwsz-3} \neq 0$  then
                msize  $\leftarrow (t \text{ and } (0-t)) || 0^4$ 

```

FIG. 16D-1

1680

```

        VirtAddr ← t and (t-1)
    else
        msize ← 64*wsiz/gsiz
        VirtAddr ← t
    endif
    vsiz ← 2*msiz*gsiz/wsiz
endcase
case of of
    W.MUL.MAT.X.I.B:
        order ← B
    W.MUL.MAT.X.I.L:
        order ← L
endcase
as ← ms ← bs ← 1
m ← LoadMemory(c,VirtAddr,msiz,order)
h ← (2*gsiz) + 7 - lgsize-(ms and bs)
r ← gsiz + (sh5||sh)
for ← 0 to wsiz-gsiz by gsiz
    q[0] ← 02*gsiz+7-lgsize
    for j ← 0 to vsiz-gsiz by gsiz
        case type of
            NONE:
                q[j+gsiz] ← q[i] + mul(gsiz,h,ms,m,i+wsiz*
                    j8..lgsize,bs,b,j)
            C:
                if (~i) & j & gsiz = 0 then
                    k ← i-(j&gsiz)+wsiz*j8..lgsize+1
                    q[j+gsiz] ← q[i] + mul(gsiz,h,ms,m,k,bs,b,j)
                else
                    k ← i+gsiz+wsiz*j8..lgsize+1
                    q[j+gsiz] ← q[j] - mul(gsiz,h,ms,m,k,bs,b,j)
                endif
            endcase
        endfor
        p ← q[vsiz]
        s ← 0h-r|| ~pr|| prr-1
        v ← ((as & ph-1)||p) + (0||s)
        if (vh..r+gsiz = (as & vr+gsiz-1)h+1-r-gsiz then
            agsiz-1+i..i ← vgsiz-1+r..r
        else
            agsiz-1+i..i ← as ? (vh||~vhgsiz-1) : 1gsiz
        endif
    endfor
    a127..wsiz ← 0
    RegWrite(rd, 128, a)
enddef

```

FIG. 16D-2

1690



Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 16E

Operation codes

W.MUL.MAT.C.F.16.B	Wide multiply matrix complex floating-point half big-endian
W.MUL.MAT.C.F.16.L	Wide multiply matrix complex floating-point little-endian
W.MUL.MAT.C.F.32.B	Wide multiply matrix complex floating-point single big-endian
W.MUL.MAT.C.F.32.L	Wide multiply matrix complex floating-point single little-endian
W.MUL.MAT.F.16.B	Wide multiply matrix floating-point half big-endian
W.MUL.MAT.F.16.L	Wide multiply matrix floating-point half little-endian
W.MUL.MAT.F.32.B	Wide multiply matrix floating-point single big-endian
W.MUL.MAT.F.32.L	Wide multiply matrix floating-point single little-endian
W.MUL.MAT.F.64.B	Wide multiply matrix floating-point double big-endian
W.MUL.MAT.F.64.L	Wide multiply matrix floating-point double little-endian

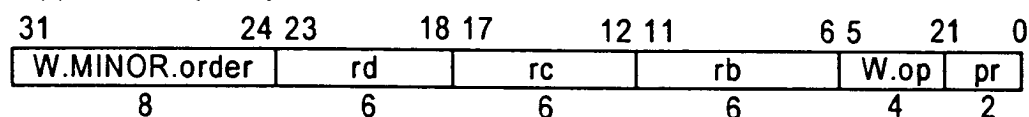
Selection

class	op	type	prec	order
wide multiply matrix	W.MUL.MAT	F	16 32 64	L B
		C.F	16 32	L B

Format

W.op.prec.order rd=rc,rb

rd=wopprecorder(rc,rb)



Pr ← log(prec) - 3

FIG. 17A

1730

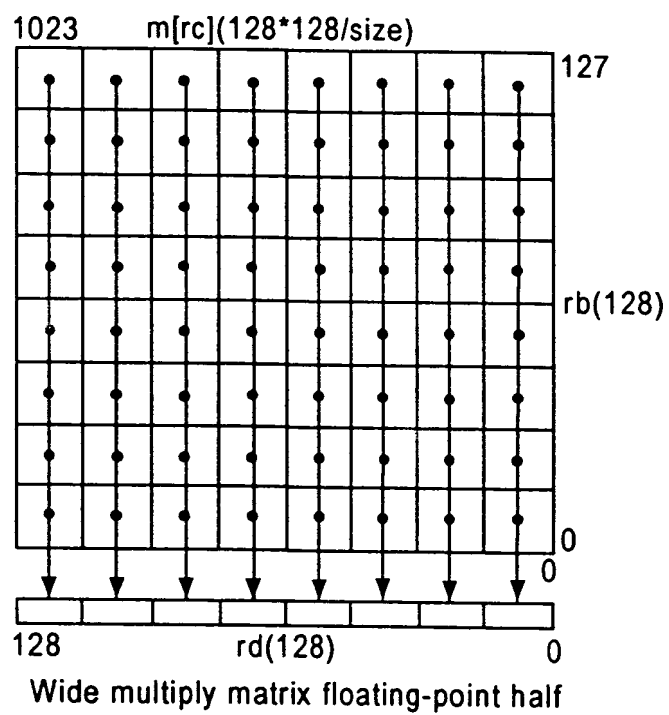


FIG. 17B

1760

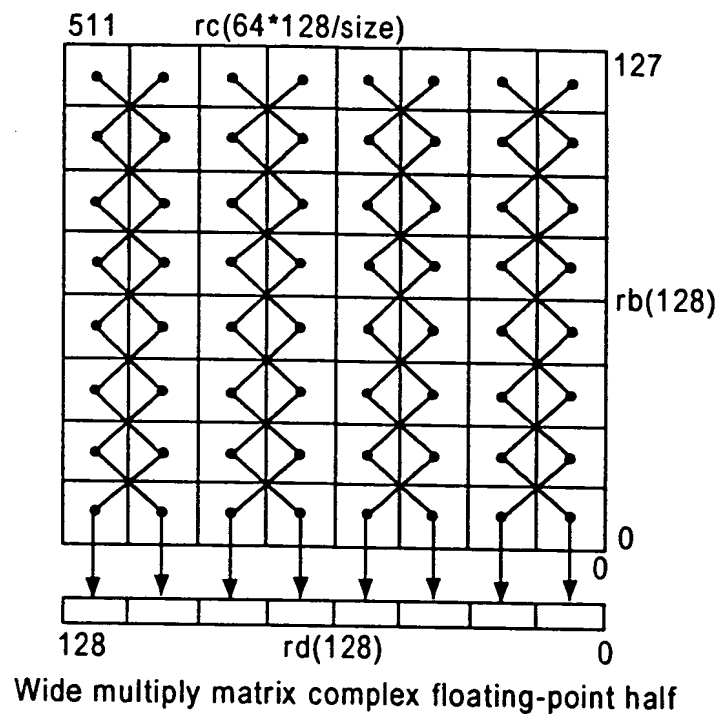


FIG. 17C

Definition

```

def mul(size,v,i,w,j) as
    mul ← fmul(F(size,vsize-1+i..i),F(size,wsiz-1+j..j))
enddef

def WideMultiplyMatrixFloatingPoint(major,op,gsiz,rd,rc,rb)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsiz ← log(gsiz)
    switch op of
        W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
            if clgsiz-4..0 ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if c3..lgsiz-3 ≠ 0 then
                wsize ← (c and (0-c)) || 04
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsiz ← log(wsize)
            if tlwsiz+6-lgsiz..lwsiz-3 ≠ 0 then
                msiz ← (t and (0-t)) || 04
                VirtAddr ← t and (t-1)
            else
                msiz ← 128*wsize/gsiz
                VirtAddr ← t
            endif
            vsiz ← msiz*gsiz/wsize
        W.MUL.MAT.C.F.16, W.MUL.MAT.C.F.32, W.MUL.MAT.C.F.64:
            if clgsiz-4..0 ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if c3..lgsiz-3 ≠ 0 then
                wsize ← (c and (0-c)) || 04
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsiz ← log(wsize)
            if tlwsiz+5-lgsiz..lwsiz-3 ≠ 0 then

```

FIG. 17D-1

1780


```

        msize ← (t and (0-t)) || 04
        VirtAddr ← t and (t-1)
    else
        msize ← 64*wsz/gsz
        VirtAddr ← t
    endif
    vsz ← 2*msz*gsz/wsz
endcase
case major of
    M.MINOR.B:
        order ← B
    M.MINOR.L:
        order ← L
endcase
m ← LoadMemory(c,VirtAddr,msz,order)
for i ← 0 to wsz-gsz by gsz
    q[0].t ← NULL
    for j ← 0 to vsz-gsz by gsz
        case op of
            W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
                q[j+gsz] ← faddq[j], mul(gsz,m,i+wsz*
                    j8..lgz+1,b,j))
            W.MUL.MAT.C.F.16, W.MUL.MAT.C.F.32,
            W.MUL.MAT.C.F.64:
                if (~i) & j & gsz = 0 then
                    k ← i-(j&gsz)+wsz*j8..lgz+1
                    q[j+gsz] ← faqq[j], mul(gsz,m,k,b,j))
                else
                    k ← i+gsz+wsz*j8..lgz+1
                    q[j+gsz] ← fsubq[j], mul(gsz,m,k,b,j))
                endif
        endcase
    endfor
    agsz-1+i..i ← q[vsz]
endfor
a127..wsz ← 0
RegWrite(rd, 128, a)
enddef

```

FIG. 17D-2

1780



Exceptions

Floating-point arithmetic
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 17E

1810

Operation codes

W.MUL.MAT.G.8.B	Wide multiply matrix Galois bytes big-endian
W.MUL.MAT.G.8.L	Wide multiply matrix Galois bytes little-endian

Selection

class	op	size	order
Multiply matrix Galois	W.MUL.MAT.G	8	B L

Format

W.op.order ra=rc,rd,rb

ra=woporder(rc,rd,rb)

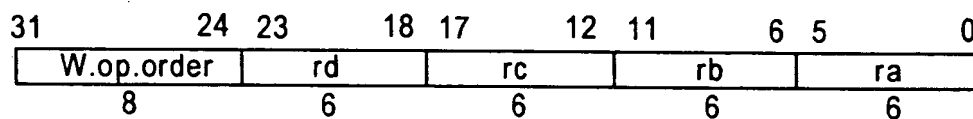


FIG. 18A

Definition

def c \leftarrow PolyMultiply(size,a,b) as
 p[0] \leftarrow 0^{2*size}
 for k \leftarrow 0 to size-1
 p[k+1] \leftarrow p[k] ^ a_k ? (0^{size-k} || b || 0^k) : 0^{2*size}
 endfor
 c \leftarrow p[size]
enddef

def c \leftarrow PolyResidue(size,a,b) as
 p[0] \leftarrow a
 for k \leftarrow size-1 to 0 by-1
 p[k-1] \leftarrow p[k] ^ p[0]_{size+k} ? (0^{size-k} || 1¹ || b || 0^k) : 0^{2*size}
 endfor
 c \leftarrow p[size]_{size-1..0}
enddef

def WideMultiplyMatrixGalois(op,gsize,rd,rc,rb,ra)
 d \leftarrow RegRead(rd, 128)
 c \leftarrow RegRead(rc, 64)
 b \leftarrow RegRead(rb, 128)
 lgsize \leftarrow log(gsize)
 if c_{lgsize-4..0} \neq 0 then
 raise AccessDisallowedByVirtualAddress
 endif
 if c_{3..lgsize-3} \neq 0 then
 wsize \leftarrow (c and (0-c)) || 0⁴
 t \leftarrow c and (c-1)
 else
 wsize \leftarrow 128
 t \leftarrow c
 endif
 lwsiz \leftarrow log(wsize)
 if t_{lwsiz+6-lgsize..lwsiz-3} \neq 0 then
 msize \leftarrow (t and (0-t)) || 0⁴
 VirtAddr \leftarrow t and (t-1)
 else
 msize \leftarrow 128*wsize/gsize
 VirtAddr \leftarrow t
 endif
 case op of
 W.MUL.MAT.G.8.B:
 order \leftarrow B
 W.MUL.MAT.G.8.L:
 order \leftarrow L
 endcase

1860

FIG. 18C-1

1860

```

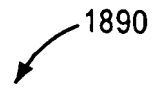
m ← LoadMemory(c, VirtAddr, msize, order)
for i ← 0 wsize-gsize by gsize
  q[0] ← 02*gsize
  for j ← 0 to vsize-gsize by gsize
    k ← i+wsize*j8..lgsize
    q[j+gsize] ← q[j] ^ PolyMultiply(gsize, mk+gsize-1..k, dj+gsize-1..j)
  endfor

  agsize-1+i..i ← PolyResidue(gsize, q[vsize], bgsize-1..0)
endfor
a127..wsize ← 0
RegWrite(ra, 128, a)
enddef

```

FIG. 18C-2

1890



Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 18D

1910

Operation codes

E.MUL.ADD.X	Ensemble multiply add extract
E.CON.X	Ensemble convolve extract

Format

E.op rd@rc,rb,ra

rd=gop(rd,rc,rb,ra)

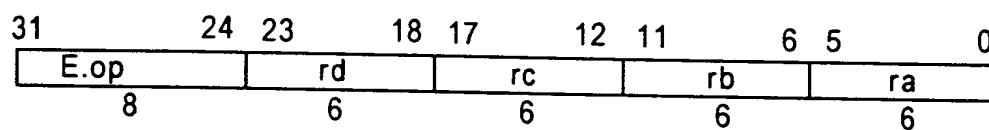


FIG. 19A

1910

Figures 19B and 20B has blank fields: should be.

fsize	dpos	x	s	n	m	l	rnd	gssp
-------	------	---	---	---	---	---	-----	------

FIG. 19B

1930

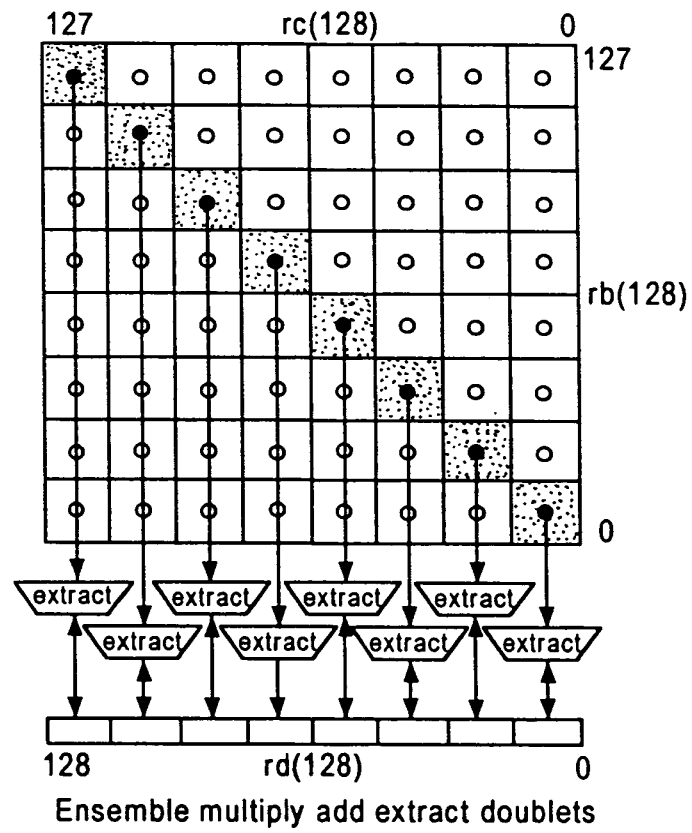
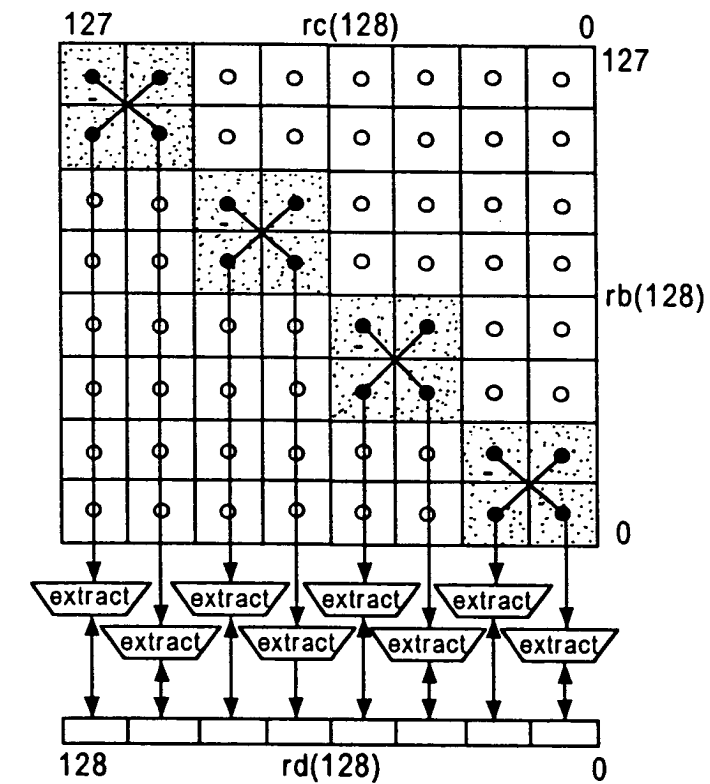


FIG. 19C



Ensemble complex multiply add extract doublets

This ensemble-multiply-add-extract instructions (E.MUL.ADD.X), when the x bit is set, multiply the low-order 64 bits of each of the rc and rb registers and produce extended (double-size) results.

FIG. 19D

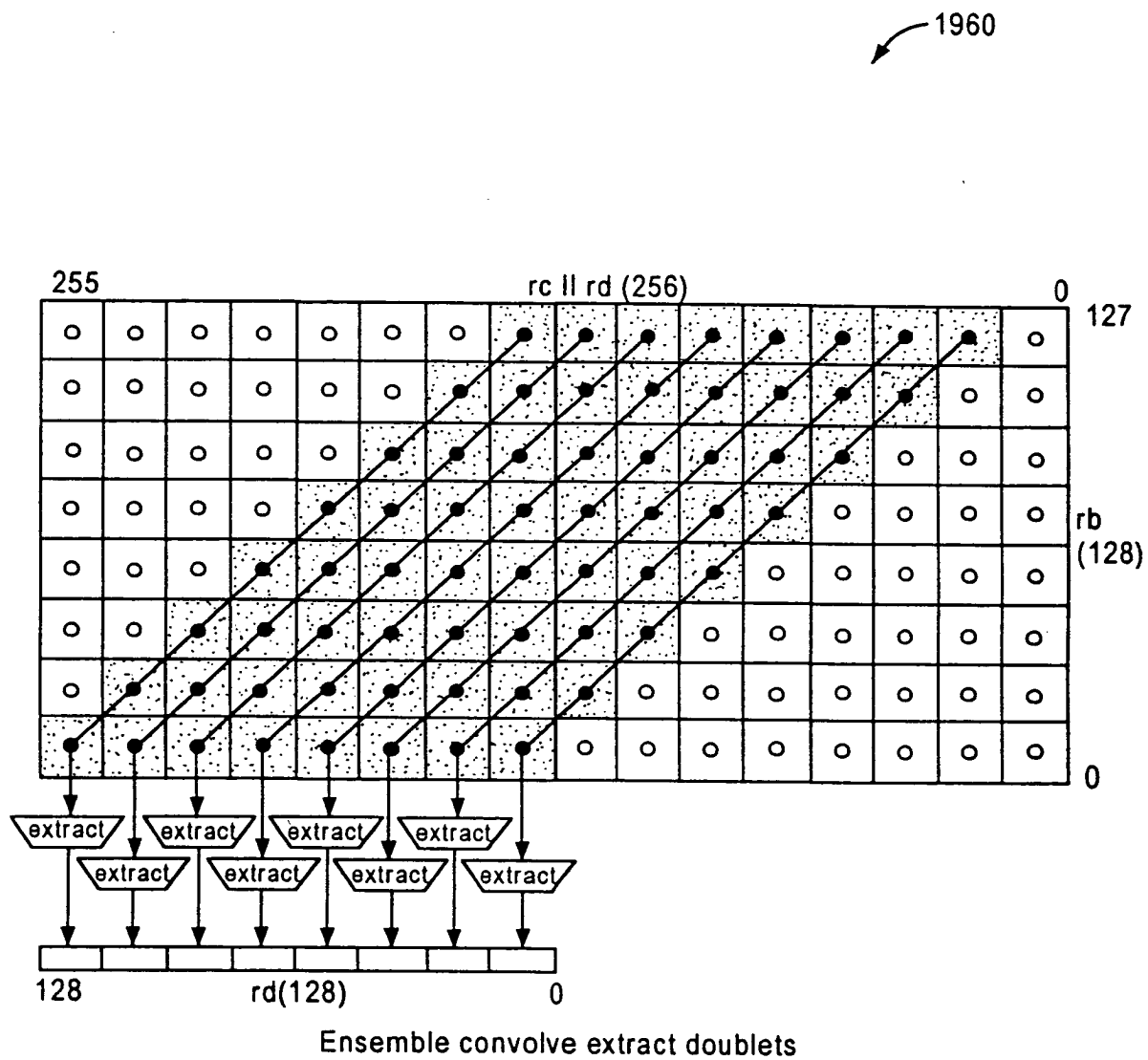


FIG. 19E

1975

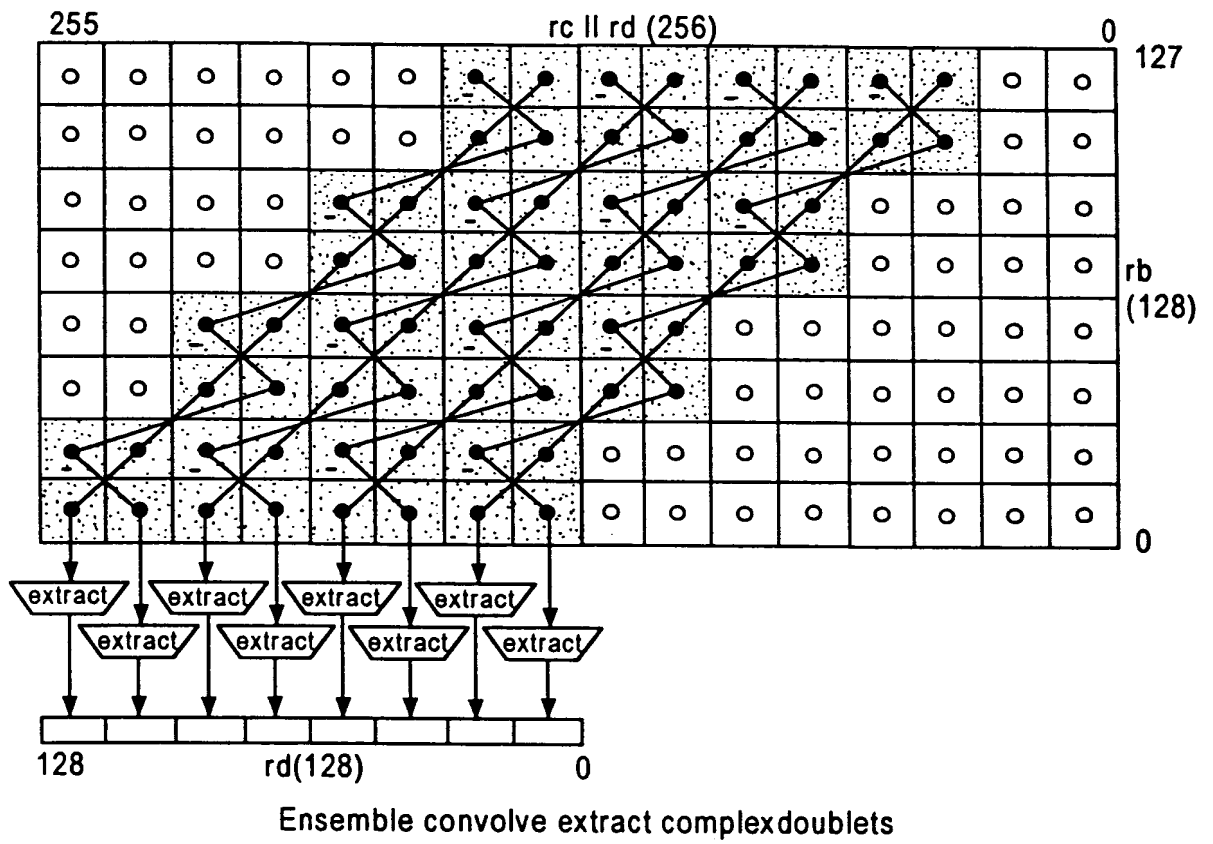


FIG. 19F

Definition

1990

```
def mul(size,h,vs,v,i,ws,w,j) as
  mul ← ((vs&vsize-1+i)h-size||vsize-1+i..i) * ((ws&wsize-1+j)h-size||wsize-1+j..j)
enddef
```

```
def EnsembleExtractInplace(op,ra,rb,rc,rd) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case b8..0 of
    0..255:
      sgsz ← 128
    256..383:
      sgsz ← 64
    384..447:
      sgsz ← 32
    448..479:
      sgsz ← 16
    480..495:
      sgsz ← 8
    496..503:
      sgsz ← 4
    504..507:
      sgsz ← 2
    508..511:
      sgsz ← 1
  endcase
  l ← a11
  m ← a12
  n ← a13
  signed ← a14
  x ← a15
  case op of
    E.CON.X:
      if (sgsz < 8) then
        gsz ← 8
      elseif (sgsz*(n-1)*(x+1) > 128 then
        gsz ← 128/(n-1)/(x+1)
      else
        gsz ← sgsz
      endif
      lgsize ← log(gsz)
      wsize ← 128/(x+1)
```

FIG. 19G-1

```

vsize ← 128
ds ← cs ← signed
bs ← signed ^ m
zs ← signed or m or n
zsize ← gsize*(x+1)
h ← (2*gszsize) + log(vsize) - lgszsize
spos ← (a8..0) and (2*gszsize-1)

```

E.MUL.ADD.X:

```

if(sgszsize < 9) then
    gsize ← 8
elseif (sgszsize*(n+1)*(x+1) > 128) then
    gsize ← 128/(n+1)/(x+1)
else
    gsize ← sgszsize
endif
ds ← signed
cs ← signed ^ m
zs ← signed or m or n
zsize ← gsize*(x+1)
h ← (2*gszsize) + n
spos ← (a8..0) and (2*gszsize-1)
endcase
dpos ← (0 || a23..16) and (zsize-1)
r ← spos
sfszsize ← (0 || a31..24) and (zsize-1)
tfszsize ← (sfszsize = 0) or ((sfszsize+dpos) > zsize) ? zsize-dpos : sfszsize
fszsize ← (tfszsize + spos > h) ? h - spos : tfszsize
if (b10..9 = Z) and not as then
    rnd ← F
else
    rnd ← b10..9
endif

```

FIG. 19G-2

```

for k ← 0 to wsize-zsize by zsize
  i ← k*gsize/zsize
  case op of
    E.CON.X:
      q[0] ← 0
      for j ← 0 to vsize-gsize by gsize
        if n then
          if (~) & j & gsize = 0 then
            q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+
              128-j,bs,b,j)
          else
            q[j+gsize] ← q[j] - mul(gsize,h,ms,i+
              128-j+2*gsize,bs,b,j)
          endif
        else
          q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+
            128-j,bs,b,j)
        endif
      endfor
      p ← q[vsize]
    E.MUL.ADD.X:
      di ← ((ds and dk+zsize-1)h-zsize-r || (dk+zsize-1..k) || 0r)
      if n then
        if (i and gsize) = 0 then
          p ← mul(gsize,h,ds,d,i,cs,c,i)-
mul(gsize,h,ds,d,i+gsize,cs,c,i+gsize)+di
        else
          p ← mul(gsize,h,ds,d,i,cs,c,i+gsize)+mul(gsize,h,ds,d,i,cs,c,i+gsize)+di
        endif
      else
        p ← mul(gsize,h,ds,d,i,cs,c,i) + di
      endif
    endif
  endcase

```

FIG. 19G-3

```

case rnd of
  N:
     $s \leftarrow 0^{h-r} \parallel \sim p_r \parallel p_r^{r-1}$ 
  Z:
     $s \leftarrow 0^{h-r} \parallel p_{h-1}^r$ 
  F:
     $s \leftarrow 0^h$ 
  C:
     $s \leftarrow 0^{h-r} \parallel 1^r$ 
endcase
 $v \leftarrow ((zs \& p_{h-1}) \parallel p) + (0 \parallel s)$ 
if ( $v_{h..r+fsz} = (zs \& v_{r+fsz-1})^{h+1-r-fsz}$ ) or not (l and (op =
EXTRACT)) then
   $w \leftarrow (zs \& v_{r+fsz-1})^{zsize-fsz-dpos} \parallel v_{fsz-1+r..r} \parallel 0^{dpos}$ 
else
   $w \leftarrow (zs ? (v_h \parallel \sim v_h^{zsize-dpos-1}) : 1^{zsize-dpos}) \parallel 0^{dpos}$ 
endif
 $z_{zsize-1-k..k} \leftarrow w$ 
endfor
RegWrite(rd, 128, z)
enddef

```

FIG. 19G-4

Operation codes

E.MUL.X	Ensemble multiply extract
E.EXTRACT	Ensemble extract
E.SCAL.ADD.X	Ensemble scale and extract

Format

E.op ra=rd,rc,rb

ra=eop(rd,rc,rb)

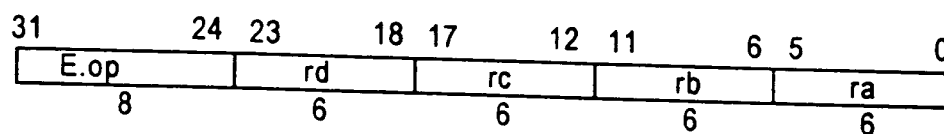


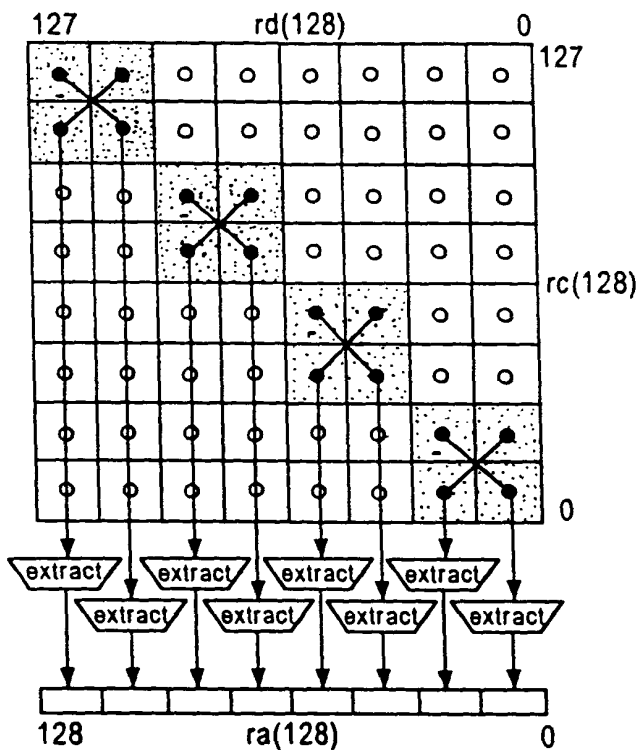
FIG. 20A

2015

Figures 19B and 20B has blank fields: should be.

fsize	dpos	x	s	n	m	l	rnd	gssp
-------	------	---	---	---	---	---	-----	------

FIG. 20B



Ensemble complex multiply extract doublets

This ensemble-multiply-extract instructions (E.MUL.X), when the x bit is set, multiply the low-order 64 bits of each of the rc and rb registers and produce extended (double-size) results.

FIG. 20D

2020

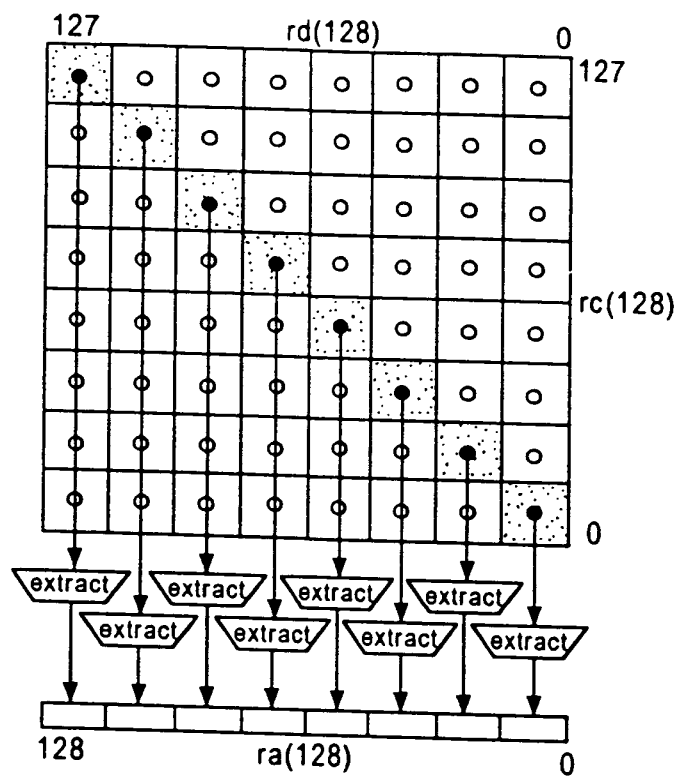
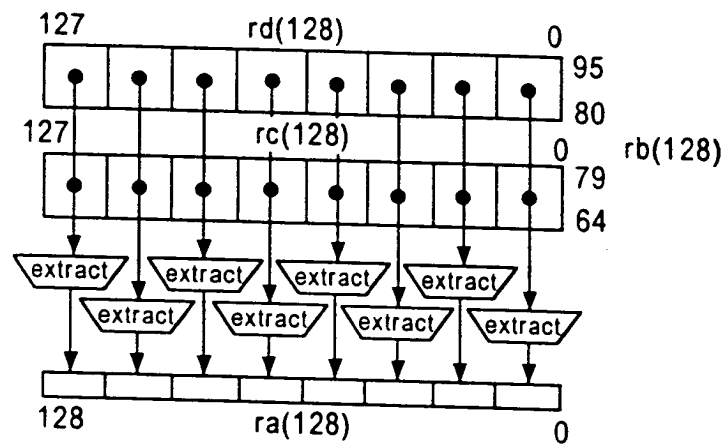


FIG. 20C

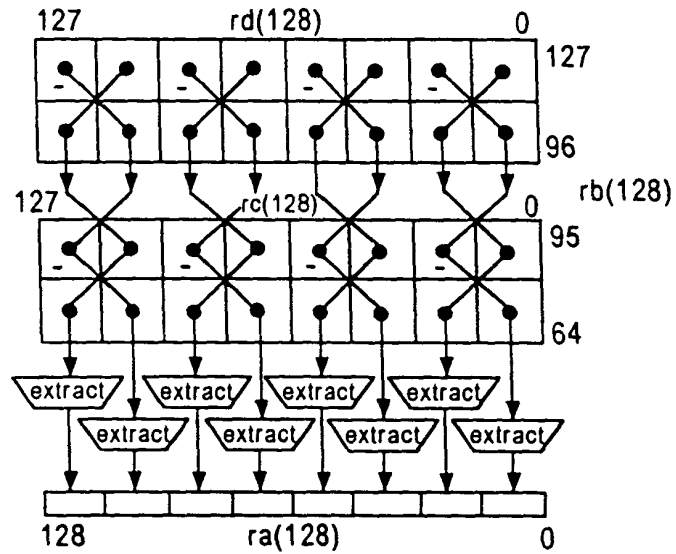
2040



Ensemble scale add extract doublets

FIG. 20E

2050

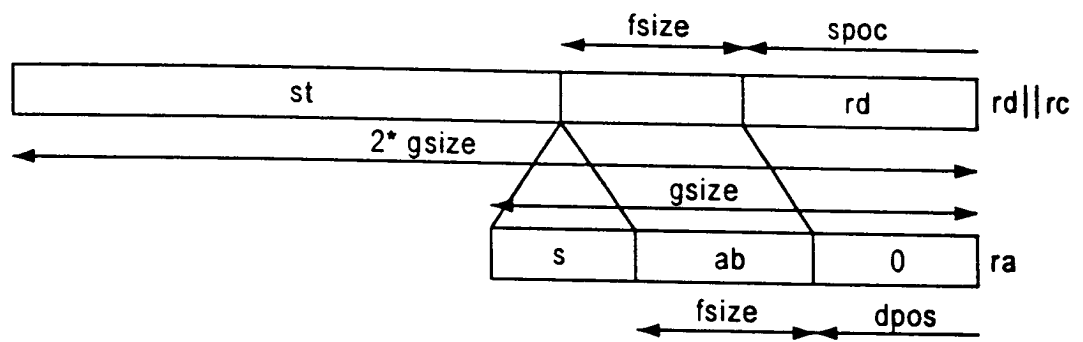


Ensemble complex scale add extract doublets

The ensemble-scale-add-extract instructions (E.SCLADD.X), when the x bit is set, multiply the low-order 64 bits of each of the rd and re registers by the rb register fields and produce extended (double-size) results.

FIG. 20F

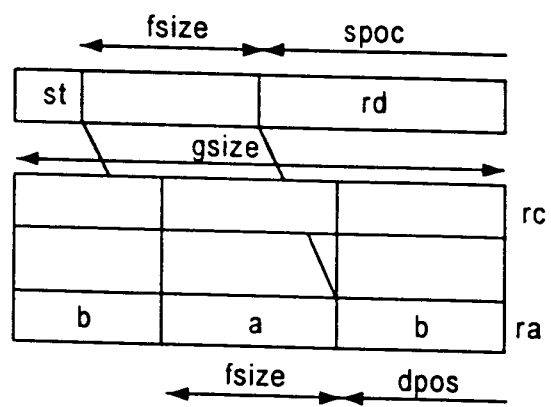
2060



Ensemble extract

FIG. 20G

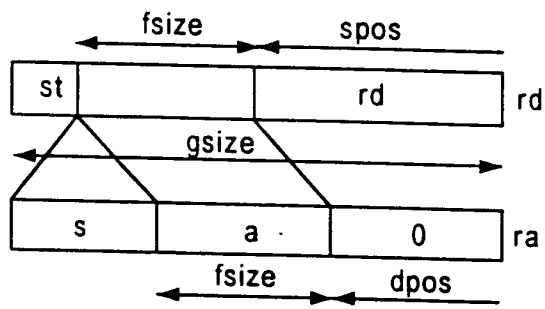
2070



Ensemble merge extract

FIG. 20H

2080



Ensemble expand extract

FIG. 20I

Definition

def mul(size,h,vs,v,i,ws,w,j) as 2090
mul ← ((vs&v_{size-1+i})^{h-size} || v_{size-1+i...i}) * ((ws&w_{size-1+j})^{h-size} || w_{size-1+j...j})
enddef

def EnsembleExtract(op,ra,rb,rc,rd) as
d ← RegRead(rd, 128)
c ← RegRead(rc, 128)
b ← RegRead(rb, 128)
case b_{8..0} of
0..255:
sgsize ← 128
256..383:
sgsize ← 64
384..447:
sgsize ← 32
448..479:
sgsize ← 16
480..495:
sgsize ← 8
496..503:
sgsize ← 4
504..507:
sgsize ← 2
508..511:
sgsize ← 1
endcase
l ← b₁₁
m ← b₁₂
n ← b₁₃
signed ← b₁₄
x ← b₁₅
case op of
E.EXTRACT:
gsize ← sgsz*2(2-(m or x))
zsize ← sgsz
h ← gsize
as ← signed
spos ← (b_{8..0}) and (gsz-1)

FIG. 20J-1

E.SCAL.ADD.X:

```

    if (sgsize < 8) then
        gsize ← 8
    elseif (sgsize*(n+1) > 32) then
        gsize ← 32/(n+1)
    else
        gsize ← sgsz
    endif
    ds ← cs ← signed
    bs ← signed ^ m
    as ← signed or m or n
    zsize ← gsize*(x+1)
    h ← (2*gsz) + 1 + n
    spos ← (b8..0) and (2*gsz-1)

```

E.MUL.X:

```

    if (sgsize < 8) then
        gsize ← 8
    elseif (sgsize*(n+1)*(x+1) > 128) then
        gsize ← 128/(n+1)/(x+1)
    else
        gsize ← sgsz
    endif
    ds ← signed
    cs ← signed ^ m
    as ← signed or m or n
    zsize ← gsize*(x+1)
    h ← (2*gsz) + n
    spos ← (b8..0) and (2*gsz-1)

```

endcase

dpos ← (0|| b23..16) and (zsize-1)

r ← spos

sfsz ← (0|| b31..24) and (zsize-1)

tfsz ← (sfsz=0) or ((sfsz+dpos) > zsize) ? zsize-dpos : sfsz

fsz ← (tfsz + spos > h) ? h - spos : tfsz

if (b10..9=Z) and not as then

rnd ← F

else

rnd ← b

endif

FIG. 20J-2

2090

```

for j ← 0 to 128-zsize by zsize
  i ← j*gsize/zsize
  case op of
    E.EXTRACT:
      if m or x then
        p ← dsize+i-1..i
      else
        p ← (d||c)gsize+i-1..i
      endif
    E.MUL.X:
      if n then
        if (i and gsize) = 0 then
          p ← mul(gsize,h,ds,d,i,cs,c,i)-
mul(gsize,h,ds,d,i+gsize,cs,c,i+gsize)
        else
          p ←
mul(gsize,h,ds,d,i,cs,c,i+gsize)+mul(gsize,h,ds,d,i,cs,c,i+gsize)
        endif
      else
        p ← mul(gsize,h,ds,d,i,cs,c,i)
      endif
    E.SCAL.ADD.X:
      if n then
        if (i and gsize) = 0 then
          p ← mul(gsize,h,ds,d,i,bs,b,64+2*gsize)
            + mul(gsize,h,cs,c,i,bs,b,64)
            - mul(gsize,h,ds,d,i+gsize,bs,b,64+3*gsize)
            - mul(gsize,h,cs,c,i+gsize,bs,b,64+gsize)
        else
          p ← mul(gsize,h,ds,d,i,bs,b,64+3*gsize)
            + mul(gsize,h,cs,c,i,bs,b,64+gsize)
            + mul(gsize,h,ds,d,i+gsize,bs,b,64+2*gsize)
            + mul(gsize,h,cs,c,i+gsize,bs,b,64)
        endif
      else
        p ← mul(gsize,h,ds,d,i,bs,b,64+gsize) + mul(gsize
            ,h,cs,c,i,bs,b,64)
      endif
    endif
  endcase

```

FIG. 20J-3

2090

```

case rnd of
  N:
     $s \leftarrow 0^{h-r} \parallel \sim p_r \parallel p_r^{r-1}$ 
  Z:
     $s \leftarrow 0^{h-r} \parallel p_{h-1}^r$ 
  F:
     $s \leftarrow 0^h$ 
  C:
     $s \leftarrow 0^{h-r} \parallel 1^r$ 
endcase
 $v \leftarrow ((as \& p_{h-1}) \parallel p) + (0 \parallel s)$ 
if ( $v_{h..r+fsz} = (as \& v_{r+fsz-1})^{h+1-r-fsz}$ ) or not (l and (op =
  E.EXTRACT)) then
   $w \leftarrow (as \& v_{r+fsz-1})^{zsize-fsz-dpos} \parallel v_{fsz-1+r..r} \parallel 0^{dpos}$ 
else
   $w \leftarrow (s ? (v_h \parallel \sim v_h^{zsize-dpos-1}) : 1^{zsize-dpos}) \parallel 0^{dpos}$ 
endif
if m and (op = E.EXTRACT) then
   $z_{size-1+j..j} \leftarrow c_{size-1+j..dpos+fsz+j} \parallel w_{dpos+fsz-1..dpos} \parallel$ 
     $c_{dpos-1+j..j}$ 
else
   $z_{size-1+j..j} \leftarrow w$ 
endif
endfor
RegWrite(ra, 128, z)
enddef

```

FIG. 20J-4

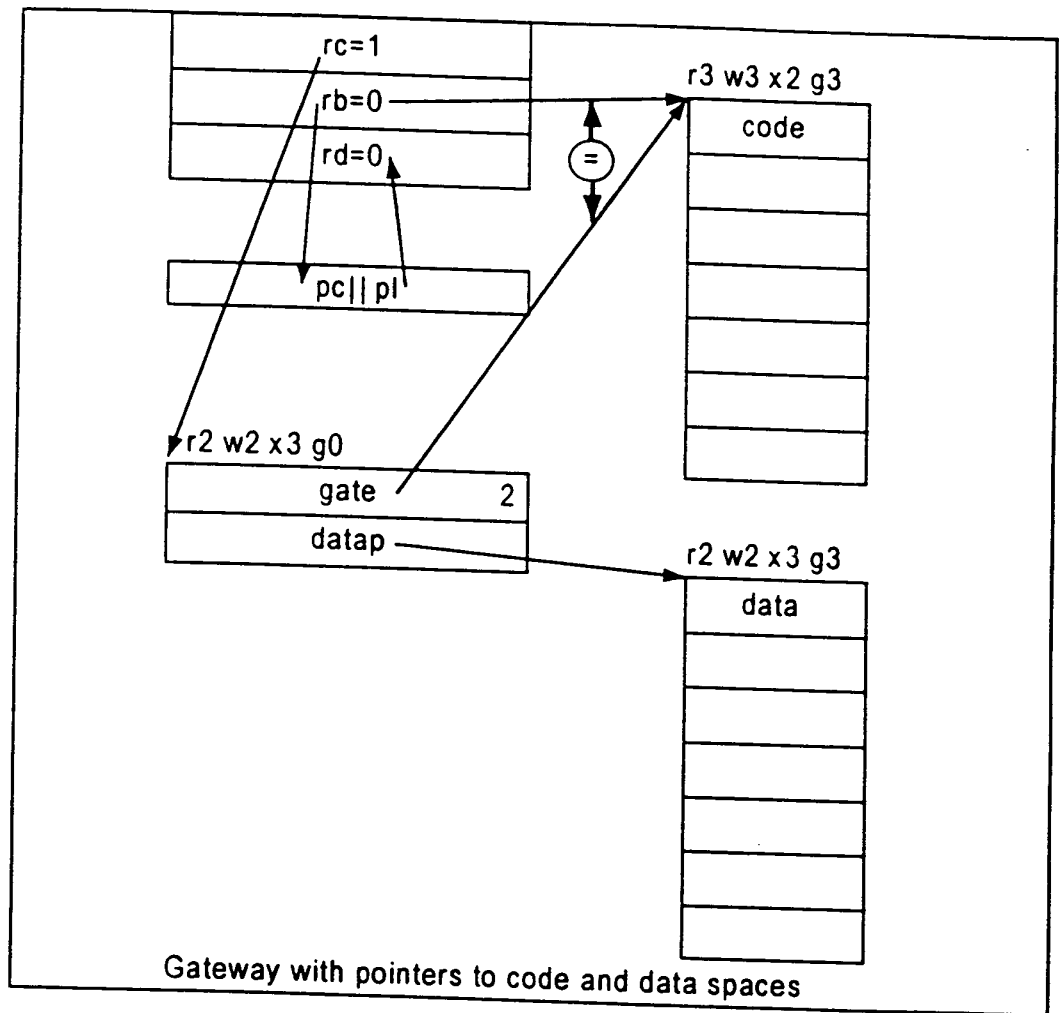


FIG. 21A

Typical dynamic-linked, inter-gateway calling sequence:

caller:

caller	AA.DDI	sp@-size	// allocate caller stack frame
	S.I.64.A	lp,sp,off	
	S.I.64.A	dp,sp,off	
	...		
	L.I.64.A	lp=dp,off	// load lp
	L.I.64.A	dp=dp,off	// load dp
	B.GATE		
	L.I.64.A	dp,sp,off	
	...(code using dp)		
	L.I.64.A	lp=sp,off	// restore original lp register
	A.ADDI	sp=size	// deallocate caller stack frame
	B	lp	// return

callee (non-leaf):

callee:	L.I.64.A	dp=dp,off	// load dp with data pointer
	S.I.64.A	sp,dp,off	
	L.I.64.A	sp=dp,off	// new stack pointer
	S.I.64.A	lp,sp,off	
	S.I.64.A	dp,sp,off	
	...(using dp)		
	L.I.64.A	dp,sp,off	
	...(code using dp)		
	L.I.64.A	lp=sp,off	// restore original lp register
	L.I.64.A	sp=sp,off	// restore original sp register
	B.DOWN	lp	

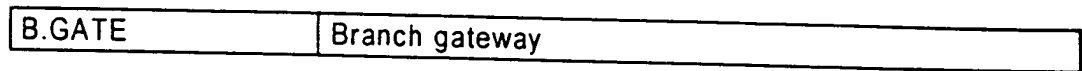
callee (leaf, no stack):

callee:	...(using dp)	
	B.DOWN	lp

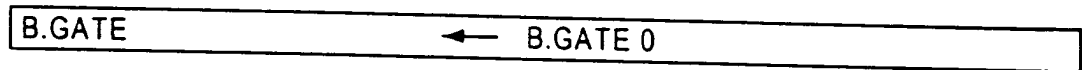
FIG. 21B

2160

Operation codes



Equivalencies



Format

B.GATE rb

bgate(rb)

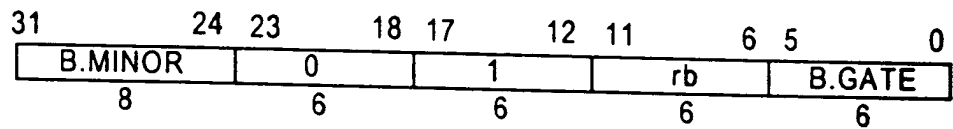


FIG. 21C

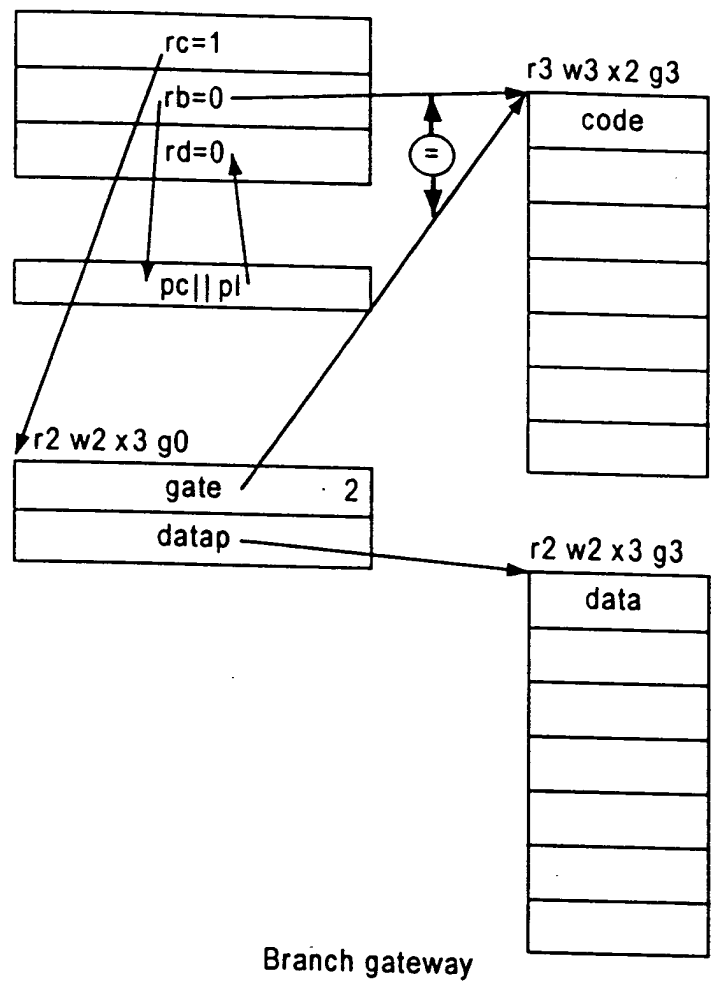


FIG. 21D

Definition

```

def BranchGateway(rd,rc,rb) as
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 64)
  if (rd ≠ 0) or (rc ≠ 1) then
    raise ReservedInstruction
  endif
  if c2..0 ≠ 0 then
    raise AccessDisallowedByVirtualAddress
  endif
  d ← ProgramCounter63..2+1 || PrivilegeLevel
  if PrivilegeLevel < b1..0 then
    m ← LoadMemoryG(c,c,64,L)
    if b ≠ m then
      raise GatewayDisallowed
    endif
    PrivilegeLevel ← b1..0
  endif
  ProgramCounter ← b63..2 || 02
  RegWrite(rd, 64, d)
  raise TakenBranch
enddef

```

FIG. 21E

Exceptions

Reserved Instruction
Gateway disallowed
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 21F

Operation codes

E.SCAL.ADD.F.16	Ensemble scale add floating-point half
E.SCAL.ADD.F.32	Ensemble scale add floating-point single
E.SCAL.ADD.F.64	Ensemble scale add floating-point double

Selection

class	op	prec
scale add	E.SCAL.ADD.F	16 32 64

Format

E.op.prec ra=rd,rc,rb

ra=eopprec(rd,rc,rb)

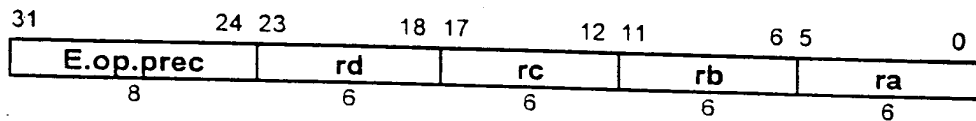


FIG. 22A

Definition

```

def EnsembleFloatingPointTernary(op,prec,rd,rc,rb,ra) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by prec
    di ← F(prec,di+prec-1..i)
    ci ← F(prec,ci+prec-1..i)
    ai ← fadd(fmul(di, F(prec,bprec-1..0)), fmul(ci, F(prec,b2*prec-1..prec)))
    ai+prec-1..i ← PackF(prec, ai, none)
  endfor
  RegWrite(ra, 128, a)
enddef

```

FIG. 22B

2310

Operation codes

G.BOOLEAN	Group boolean
-----------	---------------

Selection

operation	function (binary)	function (decimal)
d	11110000	240
c	11001100	204
b	10101010	176
d&c&b	10000000	128
(d&c) b	11101010	234
d c b	11111110	254
d?c:b	11001010	202
d^c^b	10010110	150
~d^c^b	01101001	105
0	00000000	0

Format

G.BOOLEAN rd@trc,rb,f

rd=gbooleani(rd,rc,rb,f)

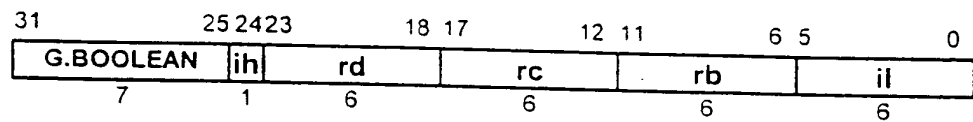


FIG. 23A

```

if f6=f5 then
  if f2=f1 then
    if f2 then
      rc ← max(trc, trb)
      rb ← min(trc, trb)
    else
      rc ← min(trc, trb)
      rb ← max(trc, trb)
    endif
    ih ← 0
    il ← 0 || f6 || f7 || f4 || f3 || f0
  else
    if f2 then
      rc ← trb
      rb ← trc
    else
      rc ← trc
      rb ← trb
    endif
    ih ← 0
    il ← 1 || f6 || f7 || f4 || f3 || f0
  endif
else
  ih ← 1
  if f6 then
    rc ← trb
    rb ← trc
    il ← f1 || f2 || f7 || f4 || f3 || f0
  else
    rc ← trc
    rb ← trb
    il ← f2 || f1 || f7 || f4 || f3 || f0
  endif
endif
endif

```

FIG. 23B

Definition

```

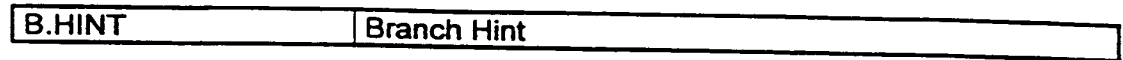
def GroupBoolean (ih,rd,rc,rb,il)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  if ih=0 then
    if il5=0 then
      f ← il3 || il4 || il4 || il2 || il1 || (rc>rb)2 || il0
    else
      f ← il3 || il4 || il4 || il2 || il1 || 0 || 1 || il0
    endif
  else
    f ← il3 || 0 || 1 || il2 || il1 || il5 || il4 || il0
  endif
  for i ← 0 to 127 by size
    ai ← f(di||ci||bi)
  endfor
  RegWrite(rd, 128, a)
enddef

```

FIG. 23C

2410 ↗

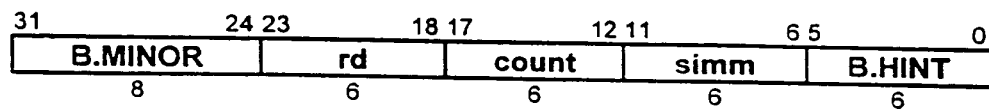
Operation codes



Format

B.HINT badd,count,rd

bhint(badd,count,rd)



simm ← badd-pc-4

FIG. 24A

2430

Definition

```
def BranchHint(rd,count,simm) as
  d ← RegRead(rd, 64)
  if (d1..0) ≠ 0 then
    raise AccessDisallowedByVirtualAddress
  endif
  FetchHint(ProgramCounter +4 + (0 || simm || 02), d63..2 || 02, count)
enddef
```

FIG. 24B

↖ 2460

Exceptions

Access disallowed by virtual address

FIG. 24C

Operation codes

E.SINK.F.16	Ensemble convert floating-point doublets from half nearest default
E.SINK.F.16C	Ensemble convert floating-point doublets from half ceiling
E.SINK.F.16.C.D	Ensemble convert floating-point doublets from half ceiling default
E.SINK.F.16.F	Ensemble convert floating-point doublets from half floor
E.SINK.F.16.F.D	Ensemble convert floating-point doublets from half floor default
E.SINK.F.16.N	Ensemble convert floating-point doublets from half nearest
E.SINK.F.16.X	Ensemble convert floating-point doublets from half exact
E.SINK.F.16.Z	Ensemble convert floating-point doublets from half zero
E.SINK.F.16.Z.D	Ensemble convert floating-point doublets from half zero default
E.SINK.F.32	Ensemble convert floating-point quadlets from single nearest default
E.SINK.F.32.C	Ensemble convert floating-point quadlets from single ceiling
E.SINK.F.32.C.D	Ensemble convert floating-point quadlets from single ceiling default
E.SINK.F.32.F	Ensemble convert floating-point quadlets from single floor
E.SINK.F.32.F.D	Ensemble convert floating-point quadlets from single floor default
E.SINK.F.32.N	Ensemble convert floating-point quadlets from single nearest
E.SINK.F.32.X	Ensemble convert floating-point quadlets from single exact
E.SINK.F.32.Z	Ensemble convert floating-point quadlets from single zero
E.SINK.F.32.Z.D	Ensemble convert floating-point quadlets from single zero default
E.SINK.F.64	Ensemble convert floating-point octlets from double nearest default
E.SINK.F.64.C	Ensemble convert floating-point octlets from double ceiling
E.SINK.F.64.C.D	Ensemble convert floating-point octlets from double ceiling default
E.SINK.F.64.F	Ensemble convert floating-point octlets from double floor
E.SINK.F.64.F.D	Ensemble convert floating-point octlets from double floor default
E.SINK.F.64.N	Ensemble convert floating-point octlets from double nearest
E.SINK.F.64.X	Ensemble convert floating-point octlets from double exact
E.SINK.F.64.Z	Ensemble convert floating-point octlets from double zero
E.SINK.F.64.Z.D	Ensemble convert floating-point octlets from double zero default
E.SINK.F.128	Ensemble convert floating-point hexlet from quad nearest default
E.SINK.F.128.C	Ensemble convert floating-point hexlet from quad ceiling
E.SINK.F.128.C.D	Ensemble convert floating-point hexlet from quad ceiling default
E.SINK.F.128.F	Ensemble convert floating-point hexlet from quad floor
E.SINK.F.128.F.D	Ensemble convert floating-point hexlet from quad floor default
E.SINK.F.128.N	Ensemble convert floating-point hexlet from quad nearest
E.SINK.F.128.X	Ensemble convert floating-point hexlet from quad exact
E.SINK.F.128.Z	Ensemble convert floating-point hexlet from quad zero
E.SINK.F.128.Z.D	Ensemble convert floating-point hexlet from quad zero default

FIG. 25A-1

Selection

	op	prec				round/trap
integer from float	SINK	16	32	64	128	NONE C F N X Z C.D F.D Z.D

Format

E.SINK.F.prec.rnd rd=rc

rd=esinkfprecnd(rc)

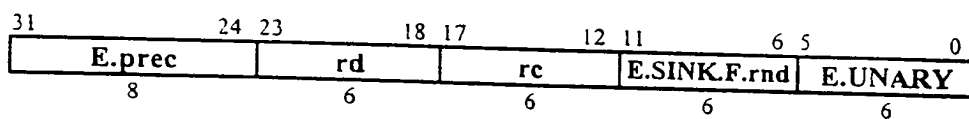


FIG. 25A-2


2530

Definition

```
def EnsembleSinkFloatingPoint(prec,round,rd,rc) as
  c ← RegRead(rc, 128)
  for i ← 0 to 128-prec by prec
    ci ← F(prec,ci+prec-1..i)
    ai+prec-1..i ← fsinkr(prec, ci, round)
  endfor
  RegWrite[rd, 128, a]
enddef
```

FIG. 25B

2560



Exceptions
Floating-point arithmetic

FIG. 25C

Definition

```
def eb ← ebits(prec) as
```

```
  case pref of
```

```
    16:
```

```
      eb ← 5
```

```
    32:
```

```
      eb ← 8
```

```
    64:
```

```
      eb ← 11
```

```
    128:
```

```
      eb ← 15
```

```
  endcase
```

```
enddef
```

```
def eb ← ebias(prec) as
```

```
  eb ← 0 || 1ebits(prec)-1
```

```
enddef
```

```
def fb ← fbits(prec) as
```

```
  fb ← prec - 1 - eb
```

```
enddef
```

```
def a ← F(prec, ai) as
```

```
  a.s ← aiprec-1
```

```
  ae ← aiprec-2..fbits(prec)
```

```
  af ← aifbits(prec)-1..0
```

```
  if ae = 1ebits(prec) then
```

```
    if af = 0 then
```

```
      a.t ← INFINITY
```

```
    elseif affbits(prec)-1 then
```

```
      a.t ← SNaN
```

```
      a.e ← -fbits(prec)
```

```
      a.f ← 1 || affbits(prec)-1..0
```

```
    else
```

```
      a.t ← QNaN
```

```
      a.e ← -fbits(prec)
```

```
      a.f ← af
```

```
    endif
```

```
  elseif ae = 0 then
```

```
    if af = 0 then
```

```
      a.t ← ZERO
```

FIG. 25D-1

```
else
    a.t ← NORM
    a.e ← 1-ebias(pec)-fbits(pec)
    a.f ← 0|| af
endif
else
    a.t ← NORM
    a.e ← ae-ebias(pec)-fbits(pec)
    a.f ← 1|| af
endif
enddef

def a ← DEFAULTQNaN as
    a.s ← 0
    a.t ← QNaN
    a.e ← -1
    a.f ← 1
endder

def a ← DEFAULTSNaN as
    a.s ← 0
    a.t ← SNaN
    a.e ← -1
    a.f ← 1
enddef
```

FIG. 25D-2

```

def fadd(a,b) as faddr(a,b,N) endder

def c ← faddr(a,b,round) as
  if a.t=NORM and b.t=NORM then
    // d,e are a,b with exponent aligned and fraction adjusted
    if a.e > b.e then
      d ← a
      e.t ← b.t
      e.s ← b.s
      e.e ← a.e
      e.f ← b.f || 0a.e-b.e
    else if a.e < b.e then
      d.t ← a.t
      d.s ← a.s
      d.e ← b.e
      d.f ← a.f || 0b.e-a.e
      e ← b
    endif
    c.t ← d.t
    c.e ← d.e
    if d.s = e.s then
      c.s ← d.s
      c.f ← d.f + e.f
    elseif d.f > e.f then
      c.s ← d.s
      c.f ← d.f - e.f
    elseif d.f < e.f then
      c.s ← e.s
      c.f ← e.f - d.f
    else
      c.s ← r=F
      c.t ← ZERO
    endif
  endif

```

FIG. 25D-3

```

// priority is given to be operand for NaN propagation
elseif (b.t=SNAN) or (b.t=QNAN) then
    c ← b
elseif (a.t=SNAN) or (a.t=QNAN) then
    c ← a
elseif a.t=ZERO and b.t=ZERO then
    c.t ← ZERO
    c.s ← (a.s and b.s) or (round=F and (a.s or b.s))
// NULL values are like zero, but do not combine with ZERO to alter sign
elseif a.t=ZERO or a.t=NULL then
    c ← b
elseif b.t=ZERO or b.t=NULL then
    c ← a
elseif a.t=INFINITY and b.t=INFINITY then
    if a.s ≠ b.s then
        c ← DEFAULTSNAN // Invalid
    else
        c ← a
    endif
elseif a.t=INFINITY then
    c ← a
elseif b.t=INFINITY then
    c ← b
else
    assert FALSE // should have covered all the cases above
endif
enddef

def b ← fneg(a) as
    b.s ← ~a.s
    b.t ← a.t
    b.e ← a.e
    b.f ← a.f
enddef

def fsub(a,b) as fsubr(a,b,N) enddef

def fsubr(a,b,round) as faddr(a,fneg(b),round) enddef

def frsub(a,b) as frsubr(a,b,N) enddef

def frsubr(a,b,round) as faddr(fneg(a),b,round) enddef

```

FIG. 25D-4

```

def c ← fcom(a,b) as
  if (a.t=SNAN) or (a.t=QNAN) or (b.t=SNAN) or (b.t=QNAN) then
    c ← U
  elseif a.t=INFINITY and b.t=INFINITY then
    if a.s ≠ b.s then
      c ← (a.s=0) ? G: L
    else
      c ← E
    endif
  elseif a.t=INFINITY then
    c ← (a.s=0) ? G: L
  elseif b.t=INFINITY then
    c ← (b.s=0) ? L
  elseif a.t=NORM and b.t=NORM then
    if a.s ≠ b.s then
      c ← (a.s=0) ? G: L
    else
      if a.e > b.e then
        af ← a.f
        bf ← b.f || 0a.e-b.e
      else
        af ← a.f || 0b.e-a.e
        bf ← b.f
      endif
      if af = bf then
        c ← E
      else
        c ← ((a.s=0) ^ (af > bf)) ? G : L
      endif
    endif
  elseif a.t=NORM then
    c ← (a.s=0) ? G: L
  elseif b.t=NORM then
    c ← (b.s=0) ? G: L
  elseif a.t=ZERO and b.t=ZERO then
    c ← E
  else
    assert FALSE // should have covered all the cases above
  endif
enddef

```

FIG. 25D-5


```

def c ← fmul(a,b) as
  if a.t=NORM and b.t=NORM then
    c.s ← a.s ^ b.s
    c.t ← NORM
    c.e ← a.e + b.e
    c.f ← a.f * b.f
  // priority is given to b operand for NaN propagation
  elseif (b.t=SNAN) or (b.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← b.t
    c.e ← b.e
    c.f ← b.f
  elseif (a.t=SNAN) or (a.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← a.t
    c.e ← a.e
    c.f ← a.f
  elseif a.t=ZERO and b.t=INFINITY then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=INFINITY and b.t=ZERO then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=ZERO or b.t=ZERO then
    c.s ← a.s ^ b.s
    c.t ← ZERO
  else
    assert FALSE // should have covered all the cases above
  endif
enddef

```

FIG. 25D-6

```

def c    fdivr(a,b) as
  if a.t=NORM and b.t=NORM then
    c.s ← a.s ^ b.s
    c.t ← NORM
    c.e ← a.e - b.e + 256
    c.f ← (a.f 0 ) / b.f
    // priority is given to b operand for NaN propagation
  elseif (b.t=SNAN) or (b.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← b.t
    c.e ← b.e
    c.f ← b.f
  elseif (a.t=SNAN) or (a.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← a.t
    c.e ← a.e
    c.f ← a.f
  elseif a.t=ZERO and b.t=INFINITY then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=INFINITY and b.t=INFINITY then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=ZERO then
    c.s ← a.s ^ b.s
    c.t ← ZERO
  elseif a.t=INFINITY then
    c.s ← a.s ^ b.s
    c.t ← INFINITY
  else
    assert FALSE // should have covered al the cases above
  endif
enddef

def msb ← findmsb(a) as
  MAXF ← 218 // Largest possible f value after matrix multiply
  for j ← 0 to MAXF
    if aMAXF-1..j = (0MAXF-1-j || 1) then
      msb ← j
    endif
  endfor
enddef

```

FIG. 25D-7

```

Def ai ← PackF(prec,a,round) as
  case a.t of
    NORM:
      msb ← findmsb(a.f)
      m ← msb-1-fbits(prec) //1sb for normal
      rdn ← -ebias(prec)-a.e-1-fbits(prec) // 1sb if a denormal
      rb ← (m > rdn) ? m : rdn
      if rb < 0 then
        aifr ← a.fmsb-1..0 || 0-rb
        eadj ← 0
      else
        case round of
          C:
            s ← 0msb-rb || (~a.s)rb
          F:
            s ← 0msb-rb || (a.s)rb
          N, NONE:
            s ← 0msb-rb || ~a.frb || a.frbb-1
          X:
            if a.frb-1..0 ≠ 0 then
              raise FloatingPointArithmetic // Inexact
            endif
            s ← 0
          Z:
            s ← 0
        endcase
      v ← (0 || a.fmsb..0) + (0 || s)
      if vmsb=1 then
        aifr ← vmsb-1..rb
        eadj ← 0
      else
        aifr ← 0fbits(prec)
        eadj ← 1
      endif
    endif
  aien ← a.e + msb - 1 + eadj + ebias(prec)
  if aien ≤ 0 then
    if round = NONE then
      ai ← a.s || 0ebits(prec) || aifr
    else
      raise FloatingPointArithmetic //Underflow
    endif
  endif

```

FIG. 25D-8

```

endif
elseif aien ≥ 1ebits(prec) then
  if round = NONE then
    //default: round-to-nearest overflow handling
    ai ← a.s || 1ebits(prec) || 0fbits(prec)
  else
    raise FloatingPointArithmetic // Overflow
  endif
else
  ai ← a.s || aienebits(prec)-1..0 || aifr
endif

SNAN:
  if round ≠ NONE then
    raise FloatingPointArithmetic //Invalid
  endif
  if -a.e < fbits(prec) then
    ai ← a.s || 1ebits(prec) || a.f-a.e-1..0 || 0fbits(prec)+a.e
  else
    lsb ← a.f-a.e-1-fbits(prec)+1..0 ≠ 0
    ai ← a.s || 1ebits(prec) || a.f-a.e-1...a.e-1-fbits(prec)+2 || 1sb
  endif
endif

QNAN:
  if -a.e < fbits(prec) then
    ai ← a.s || 1ebits(prec) || a.f-a.e-1..0 || 0fbits(prec)+a.e
  else
    1sb ← a.f-a.e-1-fbits(prec)+1..0 ≠ 0
    ai ← a.s || 1ebits(prec) || a.f-a.e-1...a.e-1-fbits(prec)+2 || 1sb
  endif
endif

ZERO:
  ai ← a.s || 0ebits(prec) || 0fbits(prec)

INFINITY:
  ai ← a.s || 1ebits(prec) || 0fbits(prec)

endcase
defdef

```

FIG. 25D-9

```

Def ai ← fsinkr(prec, a, round) as
  case a.t of
    NORM:
      msb ← findmsb(a.f)
      rb ← -a.e
      if rb ≤ 0 then
        aifr ← a.fmsb..0 || 0-rb
        aims ← msb - rb
      else
        case round of
          C,C.D:
            s ← 0msb-rb || (~ai.s)rb
          F,F.D:
            s ← 0msb-rb || (ai.s)rb
          N, NONE:
            s ← 0msb-rb || ~ai.frb || ai.frbrb-1
          X:
            if ai.frb-1..0 ≠ 0 then
              raise FloatingPointArithmetic // Inexact
            endif
            s ← 0
          Z, Z.D:
            s ← 0
        endcase
        v ← (0 || a.fmsb..0) + (0 || s)
        if vmsb = 1 then
          aims ← msb + 1 - rb
        else
          aims ← msb - rb
        endif
        aifr ← vaims..rb
      endif
    if aims > prec then
      case round of
        C.D, F.D, NONE, Z.D:
          ai ← a.s || (~a.s)prec-1
        C,F,N,X,Z:
          raise FloatingPointArithmetic // Overflow
      endcase
    end
  end

```

FIG. 25D-10

```

        elseif a.s = 0 then
            ai ← aifr
        else
            ai ← -aifr
        endif
ZERO:
    ai ← 0prec
SNAN, QNAN:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← 0prec
        C, F, N, X, Z:
            raise FloatingPoint Arithmetic // Invalid
    endcase
INFINITY:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← a.s || (~as)prec-1
        C, F, N, X, Z:
            raise FloatingPointArithmetic // Invalid
    endcase
endcase
enddef

def c    frecrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fdiv(b,a))
enddef

def c ← frsqrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fsqr(fdiv(b,a)))
enddef

```

FIG. 25D-11

```

def c ← fest(a) as
  if (a.t=NORM) then
    msb ← findmsb(a.f)
    a.e ← a.e + msb - 13
    a.f ← a.fmsb..msb-12 || 1
  else
    c ← a
  endif
enddef

def ← fsqr(a) as
  if (a.t=NORM) and (a.s=0) then
    c.s ← 0
    c.t ← NORM
    if (a.e0 = 1) then
      c.e ← (a.e-127) / 2
      c.f ← sqr(a.f || 0127)
    else
      c.e ← (a.e-128) / 2
      c.f ← sqr(a.f || 0128)
    endif
  elseif (a.t=SNAN) or (a.t=QNAN) or a.t=ZERO or ((a.t=INFINITY) and
    (a.s=0)) then
    c ← a
  elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
    c ← DEFAULTSNAN // Invalid
  else
    assert FALSE // should have covered a1 the cases above
  endif
enddef

```

FIG. 25D-12

Operation codes

G.ADD.8	Group add bytes
G.ADD.16	Group add doublets
G.ADD.32	Group add quadlets
G.ADD.64	Group add octlets
G.ADD.128	Group add hexlet
G.ADD.L.8	Group add limit signed bytes
G.ADD.L.16	Group add limit signed doublets
G.ADD.L.32	Group add limit signed quadlets
G.ADD.L.64	Group add limit signed octlets
G.ADD.L.128	Group add limit signed hexlet
G.ADD.L.U.8	Group add limit unsigned bytes
G.ADD.L.U.16	Group add limit unsigned doublets
G.ADD.L.U.32	Group add limit unsigned quadlets
G.ADD.L.U.64	Group add limit unsigned octlets
G.ADD.L.U.128	Group add limit unsigned hexlet
G.ADD.8.O	Group add signed bytes check overflow
G.ADD.16.O	Group add signed doublets check overflow
G.ADD.32.O	Group add signed quadlets check overflow
G.ADD.64.O	Group add signed octlets check overflow
G.ADD.128.O	Group add signed hexlet check overflow
G.ADD.U.8.O	Group add unsigned bytes check overflow
G.ADD.U.16.O	Group add unsigned doublets check overflow
G.ADD.U.32.O	Group add unsigned quadlets check overflow
G.ADD.U.64.O	Group add unsigned octlets check overflow
G.ADD.U.128.O	Group add unsigned hexlet check overflow

FIG. 26A

Format

G.op.size rd=rc,rb

rd=gopsize(rc,rb)

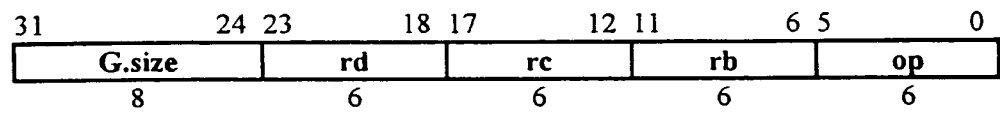


FIG. 26B

Definition

```
def Group(op,size,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    G.ADD:
      for i ← 0 to 128-size by size
        ai+size-1..i ← ci+size-1..i + bi+size-1..i
      endfor
    G.ADD.L:
      for i ← 0 to 128-size by size
        t ← (ci+size-1 || ci+size-1..i) + (bi+size-1 || bi+size-1..i)
        ai+size-1..i ← (tsize ≠ tsize-1) ? (tsize || tsize-1) : tsize-1..0
      endfor
    G.ADD.L.U:
      for i ← 0 to 128-size by size
        t ← (01 || ci+size-1..i) + (01 || bi+size-1..i)
        ai+size-1..i ← (tsize ≠ 0) ? (1size) : tsize-1..0
      endfor
    G.ADD.O:
      for i ← 0 to 128-size by size
        t ← (ci+size-1 || ci+size-1..i) + (bi+size-1 || bi+size-1..i)
        if tsize ≠ tsize-1 then
          raise FixedPointArithmetic
        endif
        ai+size-1..i ← tsize-1..0
      endfor
    G.ADD.U.O:
      for i ← 0 to 128-size by size
        t ← (01 || ci+size-1..i) + (01 || bi+size-1..i)
        if tsize ≠ 0 then
          raise FixedPointArithmetic
        endif
        ai+size-1..i ← tsize-1..0
      endfor
  endcase
  RegWrite(rd, 128, a)
enddef
```

FIG. 26C

Operation codes

G.SET.AND.E.8	Group set and equal zero bytes
G.SET.AND.E.16	Group set and equal zero doublets
G.SET.AND.E.32	Group set and equal zero quadlets
G.SET.AND.E.64	Group set and equal zero octlets
G.SET.AND.E.128	Group set and equal zero hexlet
G.SET.AND.NE.8	Group set and not equal zero bytes
G.SET.AND.NE.16	Group set and not equal zero doublets
G.SET.AND.NE.32	Group set and not equal zero quadlets
G.SET.AND.NE.64	Group set and not equal zero octlets
G.SET.AND.NE.128	Group set and not equal zero hexlet
G.SET.E.8	Group set equal bytes
G.SET.E.16	Group set equal doublets
G.SET.E.32	Group set equal quadlets
G.SET.E.64	Group set equal octlets
G.SET.E.128	Group set equal hexlet
G.SET.GE.8	Group set greater equal signed bytes
G.SET.GE.16	Group set greater equal signed doublets
G.SET.GE.32	Group set greater equal signed quadlets
G.SET.GE.64	Group set greater equal signed octlets
G.SET.GE.128	Group set greater equal signed hexlet
G.SET.GE.U.8	Group set greater equal unsigned bytes
G.SET.GE.U.16	Group set greater equal unsigned doublets
G.SET.GE.U.32	Group set greater equal unsigned quadlets
G.SET.GE.U.64	Group set greater equal unsigned octlets
G.SET.GE.U.128	Group set greater equal unsigned hexlet
G.SET.L.8	Group set signed less bytes
G.SET.L.16	Group set signed less doublets
G.SET.L.32	Group set signed less quadlets
G.SET.L.64	Group set signed less octlets
G.SET.L.128	Group set signed less hexlet
G.SET.L.U.8	Group set less unsigned bytes
G.SET.L.U.16	Group set less unsigned doublets
G.SET.L.U.32	Group set less unsigned quadlets
G.SET.L.U.64	Group set less unsigned octlets
G.SET.L.U.128	Group set less unsigned hexlet
G.SET.NE.8	Group set not equal bytes
G.SET.NE.16	Group set not equal doublets
G.SET.NE.32	Group set not equal quadlets
G.SET.NE.64	Group set not equal octlets
G.SET.NE.128	Group set not equal hexlet
G.SUB.8	Group subtract bytes
G.SUB.8.0	Group subtract signed bytes check overflow

FIG. 27A-1

G.SUB.16	Group subtract doublets
G.SUB.16.O	Group subtract signed doublets check overflow
G.SUB.32	Group subtract quadlets
G.SUB.32.O	Group subtract signed quadlets check overflow
G.SUB.64	Group subtract octlets
G.SUB.64.O	Group subtract signed octlets check overflow
G.SUB.128	Group subtract hexlet
G.SUB.128.O	Group subtract signed hexlet check overflow
G.SUB.L.8	Group subtract limit signed bytes
G.SUB.L.16	Group subtract limit signed doublets
G.SUB.L.32	Group subtract limit signed quadlets
G.SUB.L.64	Group subtract limit signed octlets
G.SUB.L.128	Group subtract limit signed hexlet
G.SUB.L.U.8	Group subtract limit unsigned bytes
G.SUB.L.U.16	Group subtract limit unsigned doublets
G.SUB.L.U.32	Group subtract limit unsigned quadlets
G.SUB.L.U.64	Group subtract limit unsigned octlets
G.SUB.L.U.128	Group subtract limit unsigned hexlet
G.SUB.U.8.O	Group subtract unsigned bytes check overflow
G.SUB.U.16.O	Group subtract unsigned doublets check overflow
G.SUB.U.32.O	Group subtract unsigned quadlets check overflow
G.SUB.U.64.O	Group subtract unsigned octlets check overflow
G.SUB.U.128.O	Group subtract unsigned hexlet check overflow

FIG. 27A-2

Format

G.op.size rd=rb,rc

rd=gopsize(rb,rc)

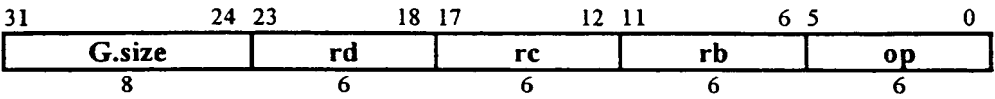


FIG. 27B

Definition

```

def GroupReversed(op,size,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    G.SUB:
      for i ← 0 to 128-size by size
        ai+size-1..i ← bi+size-1..i - ci+size-1..i
      endfor
    G.SUB.L:
      for i ← 0 to 128-size by size
        t ← (bi+size-1 || bi+size-1..i) - (ci+size-1 || ci+size-1..i)
        ai+size-1..i ← (tsize ≠ tsize-1) ? (tsize || tsize-1) : tsize-1..0
      endfor
    G.SUB.LU:
      for i ← 0 to 128-size by size
        t ← (01 || bi+size-1..i) - (01 || ci+size-1..i)
        ai+size-1..i ← (tsize ≠ 0) ? 0size: tsize-1..0
      endfor
    G.SUB.O:
      for i ← 0 to 128-size by size
        t ← (bi+size-1 || bi+size-1..i) - (ci+size-1 || ci+size-1..i)
        if (tsize ≠ tsize-1) then
          raise FixedPointArithmetic
        endif
        ai+size-1..i ← tsize-1..0
      endfor
    G.SUB.U.O:
      for i ← 0 to 128-size by size
        t ← (01 || bi+size-1..i) - (01 || ci+size-1..i)
        if (tsize ≠ 0) then
          raise FixedPointArithmetic
        endif
        ai+size-1..i ← tsize-1..0
      endfor
    G.SET.E:
      for i ← 0 to 128-size by size
        ai+size-1..i ← (bi+size-1..i = ci+size-1..i)size
      endfor
    G.SET.NE:
      for i ← 0 to 128-size by size
        ai+size-1..i ← (bi+size-1..i ≠ ci+size-1..i)size
      endfor
    G.SET.AND.E:
      for i ← 0 to 128-size by size
        ai+size-1..i ← ((bi+size-1..i and ci+size-1..i) = 0)size
      endfor
  endcase
enddef

```

FIG. 27C-1

```

G.SET.AND.NE:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((bi+size-1..i and ci+size-1..i) ≠ 0)size
  endfor
G.SET.L:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((rc = rb) ? (bi+size-1..i < 0) : (bi+size-1..i < ci+size-1..i))size
  endfor
G.SET.GE:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((rc = rb) ? (bi+size-1..i ≥ 0) : (bi+size-1..i ≥ ci+size-1..i))size
  endfor
G.SET.L.U:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((rc = rb) ? (bi+size-1..i > 0) :
      ((0 || bi+size-1..i) < (0 || ci+size-1..i)))size
  endfor
G.SET.GE.U:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((rc = rb) ? (bi+size-1..i ≤ 0) :
      ((0 || bi+size-1..i) ≥ (0 || ci+size-1..i)))size
  endfor
endcase
RegWrite(rd, 128, a)
enddef

```

FIG. 27C-2

Operation codes

E.CON.8	Ensemble convolve signed bytes
E.CON.16	Ensemble convolve signed doublets
E.CON.32	Ensemble convolve signed quadlets
E.CON.64	Ensemble convolve signed octlets
E.CON.C.8	Ensemble convolve complex bytes
E.CON.C.16	Ensemble convolve complex doublets
E.CON.C.32	Ensemble convolve complex quadlets
E.CON.M.8	Ensemble convolve mixed-signed bytes
E.CON.M.16	Ensemble convolve mixed-signed doublets
E.CON.M.32	Ensemble convolve mixed-signed quadlets
E.CON.M.64	Ensemble convolve mixed-signed octlets
E.CON.U.8	Ensemble convolve unsigned bytes
E.CON.U.16	Ensemble convolve unsigned doublets
E.CON.U.32	Ensemble convolve unsigned quadlets
E.CON.U.64	Ensemble convolve unsigned octlets
E.DIV.64	Ensemble divide signed octlets
E.DIV.U.64	Ensemble divide unsigned octlets
E.MUL.8	Ensemble multiply signed bytes
E.MUL.16	Ensemble multiply signed doublets
E.MUL.32	Ensemble multiply signed quadlets
E.MUL.64	Ensemble multiply signed octlets
E.MUL.SUM.8	Ensemble multiply sum signed bytes
E.MUL.SUM.16	Ensemble multiply sum signed doublets
E.MUL.SUM.32	Ensemble multiply sum signed quadlets
E.MUL.SUM.64	Ensemble multiply sum signed octlets
E.MUL.C.8	Ensemble complex multiply bytes
E.MUL.C.16	Ensemble complex multiply doublets
E.MUL.C.32	Ensemble complex multiply quadlets
E.MUL.M.8	Ensemble multiply mixed-signed bytes
E.MUL.M.16	Ensemble multiply mixed-signed doublets
E.MUL.M.32	Ensemble multiply mixed-signed quadlets
E.MUL.M.64	Ensemble multiply mixed-signed octlets
E.MUL.P.8	Ensemble multiply polynomial bytes
E.MUL.P.16	Ensemble multiply polynomial doublets
E.MUL.P.32	Ensemble multiply polynomial quadlets
E.MUL.P.64	Ensemble multiply polynomial octlets
E.MUL.SUM.C.8	Ensemble multiply sum complex bytes
E.MUL.SUM.C.16	Ensemble multiply sum complex doublets
E.MUL.SUM.C.32	Ensemble multiply sum complex quadlets
E.MUL.SUM.M.8	Ensemble multiply sum mixed-signed bytes
E.MUL.SUM.M.16	Ensemble multiply sum mixed-signed doublets
E.MUL.SUM.M.32	Ensemble multiply sum mixed-signed quadlets
E.MUL.SUM.M.64	Ensemble multiply sum mixed-signed octlets

FIG. 28A-1

E.MUL.SUM.U.8	Ensemble multiply sum unsigned bytes
E.MUL.SUM.U.16	Ensemble multiply sum unsigned doublets
E.MUL.SUM.U.32	Ensemble multiply sum unsigned quadlets
E.MUL.SUM.U.64	Ensemble multiply sum unsigned octlets
E.MUL.U.8	Ensemble multiply unsigned bytes
E.MUL.U.16	Ensemble multiply unsigned doublets
E.MUL.U.32	Ensemble multiply unsigned quadlets
E.MUL.U.64	Ensemble multiply unsigned octlets

FIG. 28A-2

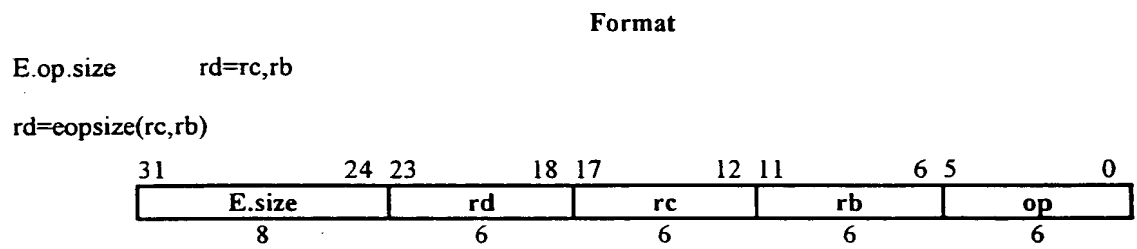


FIG. 28B

Definition

```

def mul(size,h,vs,v,i,ws,w,j) as
  mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsize-1+j)h-size || wsize-1+j..j)
enddef

def c ← PolyMultiply(size,a,b) as
  p[0] ← 02*size
  for k ← 0 to size-1
    p[k+1] ← p[k] ^ ak ? (0size-k || b || 0k) : 02*size
  endfor
  c ← p[size]
enddef

def Ensemble(op,size,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    E.MUL., E.MUL.C, EMUL.SUM, E.MUL.SUM.C, E.CON, E.CON.C, E.DIV:
      cs ← bs ← 1
    E.MUL.M., EMUL.SUM.M, E.CON.M:
      cs ← 0
      bs ← 1
    E.MUL.U., EMUL.SUM.U, E.CON.U, E.DIV.U, E.MUL.P:
      cs ← bs ← 0
  endcase
  case op of
    E.MUL, E.MUL.U, E.MUL.M:
      for i ← 0 to 64-size by size
        d2*(i+size)-1..2*i ← mul(size,2*size,cs,c,i,bs,b,i)
      endfor
    E.MUL.P:
      for i ← 0 to 64-size by size
        d2*(i+size)-1..2*i ← PolyMultiply(size,cs,c,i,bs,b,i)
      endfor
    E.MUL.C:
      for i ← 0 to 64-size by size
        if (i and size) = 0 then
          p ← mul(size,2*size,1,c,i,1,b,i) - mul(size,2*size,1,c,i+size,1,b,i+size)
        else
          p ← mul(size,2*size,1,c,i,1,b,i+size) + mul(size,2*size,1,c,i,1,b,i+size)
        endif
        d2*(i+size)-1..2*i ← p
      endfor
    E.MUL.SUM, E.MUL.SUM.U, E.MUL.SUM.M:
      p[0] ← 0128
      for i ← 0 to 128-size by size
        p[i+size] ← p[i] + mul(size,128,cs,c,i,bs,b,i)
      endfor
  endcase
enddef

```

FIG. 28C-1

```

a ← p[128]
E.MUL.SUM.C:
  p[0] ← 064
  p[size] ← 064
  for i ← 0 to 128-size by size
    if (i and size) = 0 then
      p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i)
                      - mul(size,64,1,c,i+size,1,b,i+size)
    else
      p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i+size)
                      + mul(size,64,1,c,i+size,1,b,i)
    endif
  endfor
  a ← p[128+size] || p[128]
E.CON, E.CON.U, E.CON.M:
  p[0] ← 0128
  for j ← 0 to 64-size by size
    for i ← 0 to 64-size by size
      p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i +
        mul(size,2*size,cs,c,i+64-j,bs,b,j)
    endfor
  endfor
  a ← p[64]
E.CON.C:
  p[0] ← 0128
  for j ← 0 to 64-size by size
    for i ← 0 to 64-size by size
      if ((~i) and j and size) = 0 then
        p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i +
          mul(size,2*size,1,c,i+64-j,1,b,j)
      else
        p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i -
          mul(size,2*size,1,c,i+64-j+2*size,1,b,j)
      endif
    endfor
  endfor
  a ← p[64]
E.DIV:
  if (b = 0) or ( (c = (1||063)) and (b = 164) ) then
    a ← undefined
  
```

FIG. 28C-2

```

else
     $q \leftarrow c / b$ 
     $r \leftarrow c - q * b$ 
     $a \leftarrow r_{63..0} \parallel q_{63..0}$ 
endif
E.DIV.U:
if b = 0 then
    a  $\leftarrow$  undefined
else
     $q \leftarrow (0 \parallel c) / (0 \parallel b)$ 
     $r \leftarrow c - (0 \parallel q) * (0 \parallel b)$ 
     $a \leftarrow r_{63..0} \parallel q_{63..0}$ 
endif
endcase
RegWrite(rd, 128, a)
enddef

```

FIG. 28C-3

Floating-point function Definitions

```
def eb ← ebits(prec) as
  case pref of
    16:
      eb ← 5
    32:
      eb ← 8
    64:
      eb ← 11
    128:
      eb ← 15
  endcase
enddef

def eb ← ebias(prec) as
  eb ← 0 || 1ebits(prec)-1
enddef

def fb ← fbits(prec) as
  fb ← prec - 1 - eb
enddef

def a ← F(prec, ai) as
  a.s ← aiprec-1
  ae ← aiprec-2..fbits(prec)
  af ← aifbits(prec)-1..0
  if ae = 1ebits(prec) then
    if af = 0 then
      a.t ← INFINITY
    elseif afbits(prec)-1 then
      a.t ← SNaN
      a.e ← -fbits(prec)
      a.f ← 1 || afbits(prec)-2..0
    else
      a.t ← QNaN
      a.e ← -fbits(prec)
      a.f ← af
    endif
  endif
```

FIG. 29-1

```

elseif ae = 0 then
  if af = 0 then
    a.t ← ZERO
  else
    a.t ← NORM
    a.e ← 1-ebias(prec)-fbits(prec)
    a.f ← 0 || af
  endif
else
  a.t ← NORM
  a.e ← ae-ebias(prec)-fbits(prec)
  a.f ← 1 || af
endif
enddef

def a ← DEFAULTQNaN as
  a.s ← 0
  a.t ← QNaN
  a.e ← -1
  a.f ← 1
enddef

def a ← DEFAULTSNAN as
  a.s ← 0
  a.t ← SNAN
  a.e ← -1
  a.f ← 1
enddef

def fadd(a,b) as faddr(a,b,N) enddef

def c ← faddr(a,b,round) as
  if a.t=NORM and b.t=NORM then
    // d,e are a,b with exponent aligned and fraction adjusted
    if a.e > b.e then
      d ← a
      e.t ← b.t
      e.s ← b.s
      e.e ← a.e
      e.f ← b.f || 0a.e-b.e
    else if a.e < b.e then
      d.t ← a.t
      d.s ← a.s
      d.e ← b.e
      d.f ← a.f || 0b.e-a.e
      e ← b
    end
  end
end

```

FIG. 29-2

```

endif
c.t ← d.t
c.e ← d.e
if d.s = e.s then
    c.s ← d.s
    c.f ← d.f + e.f
elseif d.f > e.f then
    c.s ← d.s
    c.f ← d.f - e.f
elseif d.f < e.f then
    c.s ← e.s
    c.f ← e.f - d.f
else
    c.s ← r=F
    c.t ← ZERO
endif
// priority is given to b operand for NaN propagation
elseif (b.t=SNAN) or (b.t=QNAN) then
    c ← b
elseif (a.t=SNAN) or (a.t=QNAN) then
    c ← a
elseif a.t=ZERO and b.t=ZERO then
    c.t ← ZERO
    c.s ← (a.s and b.s) or (round=F and (a.s or b.s))
// NULL values are like zero, but do not combine with ZERO to alter sign
elseif a.t=ZERO or a.t=NULL then
    c ← b
elseif b.t=ZERO or b.t=NULL then
    c ← a
elseif a.t=INFINITY and b.t=INFINITY then
    if a.s ≠ b.s then
        c ← DEFAULTSNAN // Invalid
    else
        c ← a
    endif
elseif a.t=INFINITY then
    c ← a
elseif b.t=INFINITY then
    c ← b
else
    assert FALSE // should have covered al the cases above
endif
enddef

def b ← fneg(a) as
    b.s ← ~a.s
    b.t ← a.t
    b.e ← a.e
    b.f ← a.f
enddef

```

FIG. 29-3


```

def fsubr(a,b,round) as faddr(a,fneg(b),round) enddef

def frsub(a,b) as frsubr(a,b,N) enddef

def frsubr(a,b,round) as faddr(fneg(a),b,round) enddef

def c ← fcom(a,b) as
  if (a.t=SNAN) or (a.t=QNAN) or (b.t=SNAN) or (b.t=QNAN) then
    c ← U
  elseif a.t=INFINITY and b.t=INFINITY then
    if a.s ≠ b.s then
      c ← (a.s=0) ? G: L
    else
      c ← E
    endif
  elseif a.t=INFINITY then
    c ← (a.s=0) ? G: L
  elseif b.t=INFINITY then
    c ← (b.s=0) ? G: L
  elseif a.t=NORM and b.t=NORM then
    if a.s ≠ b.s then
      c ← (a.s=0) ? G: L
    else
      if a.e > b.e then
        af ← a.f
        bf ← b.f || 0a.e-b.e
      else
        af ← a.f || 0b.e-a.e
        bf ← b.f
      endif
      if af = bf then
        c ← E
      else
        c ← ((a.s=0) ^ (af > bf)) ? G : L
      endif
    endif
  elseif a.t=NORM then
    c ← (a.s=0) ? G: L
  elseif b.t=NORM then
    c ← (b.s=0) ? G: L
  elseif a.t=ZERO and b.t=ZERO then
    c ← E
  else
    assert FALSE // should have covered al the cases above
  endif
enddef

```

FIG. 29-4

```

def c ← fmul(a,b) as
  if a.t=NORM and b.t=NORM then
    c.s ← a.s ^ b.s
    c.t ← NORM
    c.e ← a.e + b.e
    c.f ← a.f * b.f
  // priority is given to b operand for NaN propagation
  elseif (b.t=SNAN) or (b.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← b.t
    c.e ← b.e
    c.f ← b.f
  elseif (a.t=SNAN) or (a.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← a.t
    c.e ← a.e
    c.f ← a.f
  elseif a.t=ZERO and b.t=INFINITY then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=INFINITY and b.t=ZERO then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=ZERO or b.t=ZERO then
    c.s ← a.s ^ b.s
    c.t ← ZERO
  else
    assert FALSE // should have covered al the cases above
  endif
enddef

def c ← fdivr(a,b) as
  if a.t=NORM and b.t=NORM then
    c.s ← a.s ^ b.s
    c.t ← NORM
    c.e ← a.e - b.e + 256
    c.f ← (a.f || 0256) / b.f
  // priority is given to b operand for NaN propagation
  elseif (b.t=SNAN) or (b.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← b.t
    c.e ← b.e
    c.f ← b.f
  elseif (a.t=SNAN) or (a.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← a.t
    c.e ← a.e
    c.f ← a.f

```

FIG. 29-5

```

elseif a.t=ZERO and b.t=ZERO then
    c ← DEFAULTSNAN // Invalid
elseif a.t=INFINITY and b.t=INFINITY then
    c ← DEFAULTSNAN // Invalid
elseif a.t=ZERO then
    c.s ← a.s ^ b.s
    c.t ← ZERO
elseif a.t=INFINITY then
    c.s ← a.s ^ b.s
    c.t ← INFINITY
else
    assert FALSE // should have covered al the cases above
endif
enddef

def msb ← findmsb(a) as
    MAXF ← 218 // Largest possible f value after matrix multiply
    for j ← 0 to MAXF
        if aMAXF-1:j = (0MAXF-1-j || 1) then
            msb ← j
        endif
    endfor
enddef

def ai ← PackF(prec,a,round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            m ← msb-1-fbits(prec) // lsb for normal
            rdn ← -ebias(prec)-a.e-1-fbits(prec) // lsb if a denormal
            rb ← (m > rdn) ? m : rdn
    endcase
enddef

```

FIG. 29-6

```

if rb ≤ 0 then
    aifr ← a.fmsb-1..0 || 0-rb
    eadj ← 0
else
    case round of
        C:
            s ← 0msb-rb || (~a.s)rb
        F:
            s ← 0msb-rb || (a.s)rb
        N, NONE:
            s ← 0msb-rb || ~a.frb || a.frb-1
        X:
            if a.frb-1..0 ≠ 0 then
                raise FloatingPointArithmetic // Inexact
            endif
            s ← 0
        Z:
            s ← 0
    endcase
    v ← (0||a.fmsb..0) + (0||s)
    if vmsb = 1 then
        aifr ← vmsb-1..rb
        eadj ← 0
    else
        aifr ← 0fbits(prec)
        eadj ← 1
    endif
endif
aien ← a.e + msb - 1 + eadj + ebias(prec)
if aien ≤ 0 then
    if round = NONE then
        ai ← a.s || 0ebits(prec) || aifr
    else
        raise FloatingPointArithmetic //Underflow
    endif
elseif aien ≥ 1ebits(prec) then
    if round = NONE then
        //default: round-to-nearest overflow handling
        ai ← a.s || 1ebits(prec) || 0fbits(prec)
    else
        raise FloatingPointArithmetic //Underflow
    endif
else
    ai ← a.s || aienebits(prec)-1..0 || aifr
endif

```

FIG. 29-7

```

SNAN:
    if round ≠ NONE then
        raise FloatingPointArithmetic //Invalid
    endif
    if -a.e < fbits(prec) then
        ai ← a.s || 1ebits(prec) || a.f-a.e-1..0 || 0fbits(prec)+a.e
    else
        lsb ← a.f-a.e-1-fbits(prec)+1..0 ≠ 0
        ai ← a.s || 1ebits(prec) || a.f-a.e-1..-a.e-1-fbits(prec)+2 || lsb
    endif
QNaN:
    if -a.e < fbits(prec) then
        ai ← a.s || 1ebits(prec) || a.f-a.e-1..0 || 0fbits(prec)+a.e
    else
        lsb ← a.f-a.e-1-fbits(prec)+1..0 ≠ 0
        ai ← a.s || 1ebits(prec) || a.f-a.e-1..-a.e-1-fbits(prec)+2 || lsb
    endif
ZERO:
    ai ← a.s || 0ebits(prec) || 0fbits(prec)
INFINITY:
    ai ← a.s || 1ebits(prec) || 0fbits(prec)
endcase
defdef

def ai ← fsinkr(prec, a, round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            rb ← -a.e
            if rb ≤ 0 then
                aifr ← a.fmsb..0 || 0-rb
                aims ← msb - rb
            else
                case round of
                    C, C.D:
                        s ← 0msb-rb || (~ai.s)rb
                    F, F.D:
                        s ← 0msb-rb || (ai.s)rb
                    N, NONE:
                        s ← 0msb-rb || ~ai.frb || ai.frb-1
                    X:
                        if ai.frb-1..0 ≠ 0 then
                            raise FloatingPointArithmetic // Inexact
                        endif
                        s ← 0
                    Z, Z.D:
                        s ← 0

```

FIG. 29-8

```

        endcase
        v ← (0||a.fmsb..0) + (0||s)
        if vmsb = 1 then
            aims ← msb + 1 - rb
        else
            aims ← msb - rb
        endif
        aifr ← vaims..rb
    endif
    if aims > prec then
        case round of
            C.D, F.D, NONE, Z.D:
                ai ← a.s || (~as)prec-1

                C, F, N, X, Z:
                    raise FloatingPointArithmetic // Overflow
        endcase
    elseif a.s = 0 then
        ai ← aifr
    else
        ai ← -aifr
    endif
ZERO:
    ai ← 0prec
SNAN, QNAN:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← 0prec
        C, F, N, X, Z:
            raise FloatingPointArithmetic // Invalid
    endcase
INFINITY:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← a.s || (~as)prec-1
        C, F, N, X, Z:
            raise FloatingPointArithmetic // Invalid
    endcase
endcase
enddef

def c ← frecrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fdiv(b,a))
enddef

```

FIG. 29-9

```

def c ← frsqrest(a) as
  b.s ← 0
  b.t ← NORM
  b.e ← 0
  b.f ← 1
  c ← fest(fsqr(fdiv(b,a)))
enddef

def c ← fest(a) as
  if (a.t=NORM) then
    msb ← findmsb(a.f)
    a.e ← a.e + msb - 13
    a.f ← a.fmsb..msb-12 || 1
  else
    c ← a
  endif
enddef

def c ← fsqr(a) as
  if (a.t=NORM) and (a.s=0) then
    c.s ← 0
    c.t ← NORM
    if (a.e0 = 1) then
      c.e ← (a.e-127) / 2
      c.f ← sqr(a.f || 0127)
    else
      c.e ← (a.e-128) / 2
      c.f ← sqr(a.f || 0128)
    endif
  elseif (a.t=SNAN) or (a.t=QNaN) or a.t=ZERO or ((a.t=INFINITY) and (a.s=0)) then
    c ← a
  elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
    c ← DEFAULTSNAN // Invalid
  else
    assert FALSE // should have covered al the cases above
  endif
enddef

```

FIG. 29-10

Operation codes

E.ADD.F.16	Ensemble add floating-point half
E.ADD.F.16.C	Ensemble add floating-point half ceiling
E.ADD.F.16.F	Ensemble add floating-point half floor
E.ADD.F.16.N	Ensemble add floating-point half nearest
E.ADD.F.16.X	Ensemble add floating-point half exact
E.ADD.F.16.Z	Ensemble add floating-point half zero
E.ADD.F.32	Ensemble add floating-point single
E.ADD.F.32.C	Ensemble add floating-point single ceiling
E.ADD.F.32.F	Ensemble add floating-point single floor
E.ADD.F.32.N	Ensemble add floating-point single nearest
E.ADD.F.32.X	Ensemble add floating-point single exact
E.ADD.F.32.Z	Ensemble add floating-point single zero
E.ADD.F.64	Ensemble add floating-point double
E.ADD.F.64.C	Ensemble add floating-point double ceiling
E.ADD.F.64.F	Ensemble add floating-point double floor
E.ADD.F.64.N	Ensemble add floating-point double nearest
E.ADD.F.64.X	Ensemble add floating-point double exact
E.ADD.F.64.Z	Ensemble add floating-point double zero
E.ADD.F.128	Ensemble add floating-point quad
E.ADD.F.128.C	Ensemble add floating-point quad ceiling
E.ADD.F.128.F	Ensemble add floating-point quad floor
E.ADD.F.128.N	Ensemble add floating-point quad nearest
E.ADD.F.128.X	Ensemble add floating-point quad exact
E.ADD.F.128.Z	Ensemble add floating-point quad zero
E.DIV.F.16	Ensemble divide floating-point half
E.DIV.F.16.C	Ensemble divide floating-point half ceiling
E.DIV.F.16.F	Ensemble divide floating-point half floor
E.DIV.F.16.N	Ensemble divide floating-point half nearest
E.DIV.F.16.X	Ensemble divide floating-point half exact
E.DIV.F.16.Z	Ensemble divide floating-point half zero
E.DIV.F.32	Ensemble divide floating-point single
E.DIV.F.32.C	Ensemble divide floating-point single ceiling
E.DIV.F.32.F	Ensemble divide floating-point single floor
E.DIV.F.32.N	Ensemble divide floating-point single nearest
E.DIV.F.32.X	Ensemble divide floating-point single exact
E.DIV.F.32.Z	Ensemble divide floating-point single zero
E.DIV.F.64	Ensemble divide floating-point double

FIG. 30A-1

E.DIV.F.64.C	Ensemble divide floating-point double ceiling
E.DIV.F.64.F	Ensemble divide floating-point double floor
E.DIV.F.64.N	Ensemble divide floating-point double nearest
E.DIV.F.64.X	Ensemble divide floating-point double exact
E.DIV.F.64.Z	Ensemble divide floating-point double zero
E.DIV.F.128	Ensemble divide floating-point quad
E.DIV.F.128.C	Ensemble divide floating-point quad ceiling
E.DIV.F.128.F	Ensemble divide floating-point quad floor
E.DIV.F.128.N	Ensemble divide floating-point quad nearest
E.DIV.F.128.X	Ensemble divide floating-point quad exact
E.DIV.F.128.Z	Ensemble divide floating-point quad zero
E.MUL.C.F.16	Ensemble multiply complex floating-point half
E.MUL.C.F.32	Ensemble multiply complex floating-point single
E.MUL.C.F.64	Ensemble multiply complex floating-point double
E.MUL.F.16	Ensemble multiply floating-point half
E.MUL.F.16.C	Ensemble multiply floating-point half ceiling
E.MUL.F.16.F	Ensemble multiply floating-point half floor
E.MUL.F.16.N	Ensemble multiply floating-point half nearest
E.MUL.F.16.X	Ensemble multiply floating-point half exact
E.MUL.F.16.Z	Ensemble multiply floating-point half zero
E.MUL.F.32	Ensemble multiply floating-point single
E.MUL.F.32.C	Ensemble multiply floating-point single ceiling
E.MUL.F.32.F	Ensemble multiply floating-point single floor
E.MUL.F.32.N	Ensemble multiply floating-point single nearest
E.MUL.F.32.X	Ensemble multiply floating-point single exact
E.MUL.F.32.Z	Ensemble multiply floating-point single zero
E.MUL.F.64	Ensemble multiply floating-point double
E.MUL.F.64.C	Ensemble multiply floating-point double ceiling
E.MUL.F.64.F	Ensemble multiply floating-point double floor
E.MUL.F.64.N	Ensemble multiply floating-point double nearest
E.MUL.F.64.X	Ensemble multiply floating-point double exact
E.MUL.F.64.Z	Ensemble multiply floating-point double zero
E.MUL.F.128	Ensemble multiply floating-point quad
E.MUL.F.128.C	Ensemble multiply floating-point quad ceiling
E.MUL.F.128.F	Ensemble multiply floating-point quad floor
E.MUL.F.128.N	Ensemble multiply floating-point quad nearest
E.MUL.F.128.X	Ensemble multiply floating-point quad exact
E.MUL.F.128.Z	Ensemble multiply floating-point quad zero

FIG. 30A-2

Selection

class	op	prec				round/trap
add	EADDF	16	32	64	128	NONE C F N X Z
divide	EDIVF	16	32	64	128	NONE C F N X Z
multiply	EMULF	16	32	64	128	NONE C F N X Z
complex multiply	EMUL.CF	16	32	64		NONE

Format

E.op.prec.round rd=rc,rb

rd=eopprecround(rc,rb)

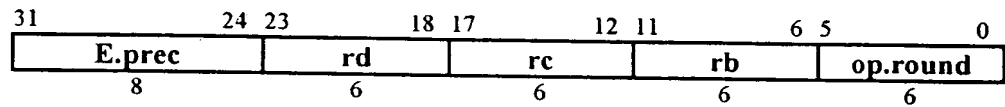


FIG. 30B

Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,vsize-1+i..i),F(size,wsizel-1+j..j))
enddef

def EnsembleFloatingPoint(op,prec,round,ra,rb,rc) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        ci ← F(prec,ci+prec-1..i)
        bi ← F(prec,bi+prec-1..i)
        case op of
            E.ADD.F:
                ai ← faddr(ci,bi,round)
            E.MUL.F:
                ai ← fmul(ci,bi)
            E.MUL.C.F:
                if (i and prec) then
                    ai ← fadd(mul(prec,c,i,b,i-prec), mul(prec,c,i-prec,b,i))
                else
                    ai ← fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec))
                endif
            E.DIV.F.:
                ai ← fdiv(ci,bi)
        endcase
        ai+prec-1..i ← PackF(prec, ai, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

FIG. 30C

Operation codes

E.SUB.F.16	Ensemble subtract floating-point half
E.SUB.F.16.C	Ensemble subtract floating-point half ceiling
E.SUB.F.16.F	Ensemble subtract floating-point half floor
E.SUB.F.16.N	Ensemble subtract floating-point half nearest
E.SUB.F.16.Z	Ensemble subtract floating-point half zero
E.SUB.F.16.X	Ensemble subtract floating-point half exact
E.SUB.F.32	Ensemble subtract floating-point single
E.SUB.F.32.C	Ensemble subtract floating-point single ceiling
E.SUB.F.32.F	Ensemble subtract floating-point single floor
E.SUB.F.32.N	Ensemble subtract floating-point single nearest
E.SUB.F.32.Z	Ensemble subtract floating-point single zero
E.SUB.F.32.X	Ensemble subtract floating-point single exact
E.SUB.F.64	Ensemble subtract floating-point double
E.SUB.F.64.C	Ensemble subtract floating-point double ceiling
E.SUB.F.64.F	Ensemble subtract floating-point double floor
E.SUB.F.64.N	Ensemble subtract floating-point double nearest
E.SUB.F.64.Z	Ensemble subtract floating-point double zero
E.SUB.F.64.X	Ensemble subtract floating-point double exact
E.SUB.F.128	Ensemble subtract floating-point quad
E.SUB.F.128.C	Ensemble subtract floating-point quad ceiling
E.SUB.F.128.F	Ensemble subtract floating-point quad floor
E.SUB.F.128.N	Ensemble subtract floating-point quad nearest
E.SUB.F.128.Z	Ensemble subtract floating-point quad zero
E.SUB.F.128.X	Ensemble subtract floating-point quad exact

FIG. 31A

Selection

class	op	prec	round/trap
set	SET. E LG L GE	16 32 64 128	NONE X
subtract	SUB	16 32 64 128	NONE C F N X Z

Format

E.op.prec.round rd=rb,rc

rd=eopprecround(rb,rc)

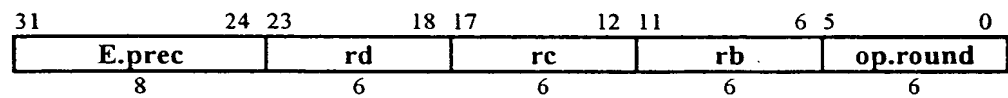


FIG. 31B

Definition

```
def EnsembleReversedFloatingPoint(op,prec,round,rd,rc,rb) as
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by prec
    ci ← F(prec,ci+prec-1..i)
    bi ← F(prec,bi+prec-1..i)
    ai ← frsubr(ci,-bi, round)
    ai+prec-1..i ← PackF(prec, ai, round)
  endfor
  RegWrite(rd, 128, a)
enddef
```

FIG. 31C

Operation codes

X.COMPRESS.2	Crossbar compress signed pecks
X.COMPRESS.4	Crossbar compress signed nibbles
X.COMPRESS.8	Crossbar compress signed bytes
X.COMPRESS.16	Crossbar compress signed doublets
X.COMPRESS.32	Crossbar compress signed quadlets
X.COMPRESS.64	Crossbar compress signed octlets
X.COMPRESS.128	Crossbar compress signed hexlet
X.COMPRESS.U.2	Crossbar compress unsigned pecks
X.COMPRESS.U.4	Crossbar compress unsigned nibbles
X.COMPRESS.U.8	Crossbar compress unsigned bytes
X.COMPRESS.U.16	Crossbar compress unsigned doublets
X.COMPRESS.U.32	Crossbar compress unsigned quadlets
X.COMPRESS.U.64	Crossbar compress unsigned octlets
X.COMPRESS.U.128	Crossbar compress unsigned hexlet
X.EXPAND.2	Crossbar expand signed pecks
X.EXPAND.4	Crossbar expand signed nibbles
X.EXPAND.8	Crossbar expand signed bytes
X.EXPAND.16	Crossbar expand signed doublets
X.EXPAND.32	Crossbar expand signed quadlets
X.EXPAND.64	Crossbar expand signed octlets
X.EXPAND.128	Crossbar expand signed hexlet
X.EXPAND.U.2	Crossbar expand unsigned pecks
X.EXPAND.U.4	Crossbar expand unsigned nibbles
X.EXPAND.U.8	Crossbar expand unsigned bytes
X.EXPAND.U.16	Crossbar expand unsigned doublets
X.EXPAND.U.32	Crossbar expand unsigned quadlets
X.EXPAND.U.64	Crossbar expand unsigned octlets
X.EXPAND.U.128	Crossbar expand unsigned hexlet
X.ROTL.2	Crossbar rotate left pecks
X.ROTL.4	Crossbar rotate left nibbles
X.ROTL.8	Crossbar rotate left bytes
X.ROTL.16	Crossbar rotate left doublets
X.ROTL.32	Crossbar rotate left quadlets
X.ROTL.64	Crossbar rotate left octlets
X.ROTL.128	Crossbar rotate left hexlet
X.ROTR.2	Crossbar rotate right pecks
X.ROTR.4	Crossbar rotate right nibbles
X.ROTR.8	Crossbar rotate right bytes
X.ROTR.16	Crossbar rotate right doublets

FIG. 32A-1

X.ROTR.32	Crossbar rotate right quadlets
X.ROTR.64	Crossbar rotate right octlets
X.ROTR.128	Crossbar rotate right hexlet
X.SHL.2	Crossbar shift left pecks
X.SHL.2.O	Crossbar shift left signed pecks check overflow
X.SHL.4	Crossbar shift left nibbles
X.SHL.4.O	Crossbar shift left signed nibbles check overflow
X.SHL.8	Crossbar shift left bytes
X.SHL.8.O	Crossbar shift left signed bytes check overflow
X.SHL.16	Crossbar shift left doublets
X.SHL.16.O	Crossbar shift left signed doublets check overflow
X.SHL.32	Crossbar shift left quadlets
X.SHL.32.O	Crossbar shift left signed quadlets check overflow
X.SHL.64	Crossbar shift left octlets
X.SHL.64.O	Crossbar shift left signed octlets check overflow
X.SHL.128	Crossbar shift left hexlet
X.SHL.128.O	Crossbar shift left signed hexlet check overflow
X.SHL.U.2.O	Crossbar shift left unsigned pecks check overflow
X.SHL.U.4.O	Crossbar shift left unsigned nibbles check overflow
X.SHL.U.8.O	Crossbar shift left unsigned bytes check overflow
X.SHL.U.16.O	Crossbar shift left unsigned doublets check overflow
X.SHL.U.32.O	Crossbar shift left unsigned quadlets check overflow
X.SHL.U.64.O	Crossbar shift left unsigned octlets check overflow
X.SHL.U.128.O	Crossbar shift left unsigned hexlet check overflow
X.SHR.2	Crossbar signed shift right pecks
X.SHR.4	Crossbar signed shift right nibbles
X.SHR.8	Crossbar signed shift right bytes
X.SHR.16	Crossbar signed shift right doublets
X.SHR.32	Crossbar signed shift right quadlets
X.SHR.64	Crossbar signed shift right octlets
X.SHR.128	Crossbar signed shift right hexlet
X.SHR.U.2	Crossbar shift right unsigned pecks
X.SHR.U.4	Crossbar shift right unsigned nibbles
X.SHR.U.8	Crossbar shift right unsigned bytes
X.SHR.U.16	Crossbar shift right unsigned doublets
X.SHR.U.32	Crossbar shift right unsigned quadlets
X.SHR.U.64	Crossbar shift right unsigned octlets
X.SHR.U.128	Crossbar shift right unsigned hexlet

FIG. 32A-2

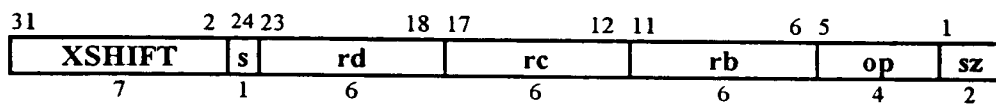
Selection

class	op				size							
precision	EXPAND	EXPAND.U			2	4	8	16	32	64	128	
	COMPRESS	COMPRESS.U										
shift	ROTR	ROTL	SHR	SHL	2	4	8	16	32	64	128	
	SHL.O	SHL.U.O	SHR.U									

Format

X.op.size rd=rc,rb

rd=xopsize(rc,rb)



lsize \leftarrow log(size)

s \leftarrow lsize2

sz \leftarrow lsize1..0

FIG. 32B

Definition

```
def Crossbar(op,size,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  shift ← b and (size-1)
  case op5..2 || 02 of
    X.COMPRESS:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+hsize-1..i ← ci+i+shift+hsize-1..i+i+shift
        else
          ai+hsize-1..i ← cshift-hsizei+i+size-1 || ci+i+size-1..i+i+shift
        endif
      endfor
      a127..64 ← 0
    X.COMPRESS.U:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+hsize-1..i ← ci+i+shift+hsize-1..i+i+shift
        else
          ai+hsize-1..i ← 0shift-hsize || ci+i+size-1..i+i+shift
        endif
      endfor
      a127..64 ← 0
    X.EXPAND:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+i+size-1..i ← chsize-shifti+hsize-1 || ci+hsize-1..i || 0shift
        else
          ai+i+size-1..i ← ci+size-shift-1..i || 0shift
        endif
      endfor
```

FIG. 32C-1

X.EXPAND.U:

```
    hsize ← size/2
    for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
             $a_{i+i+size-1..i+i} \leftarrow 0^{hsize-shift} \parallel c_{i+hsize-1..i} \parallel 0^{shift}$ 
        else
             $a_{i+i+size-1..i+i} \leftarrow c_{i+size-shift-1..i} \parallel 0^{shift}$ 
        endif
    endfor
```

X.ROTL:

```
    for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel c_{i+size-1..i+size-1-shift}$ 
    endfor
```

X.ROTR:

```
    for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow c_{i+shift-1..i} \parallel c_{i+size-1..i+shift}$ 
    endfor
```

X.SHL:

```
    for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0^{shift}$ 
    endfor
```

X.SHL.O:

```
    for i ← 0 to 128-size by size
        if  $c_{i+size-1..i+size-1-shift} \neq c_{i+size-1-shift}^{shift+1}$  then
            raise FixedPointArithmetic
        endif
         $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0^{shift}$ 
    endfor
```

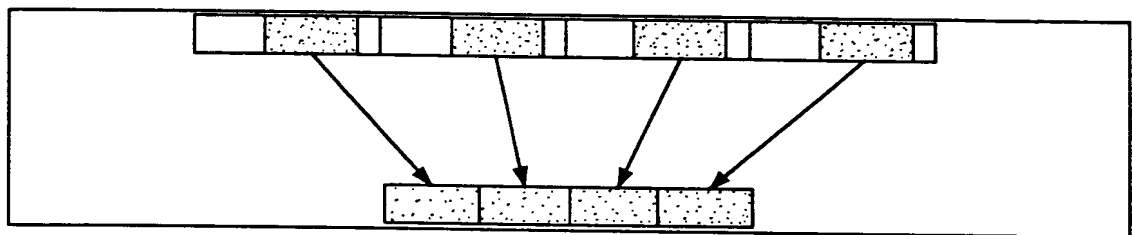
FIG. 32C-2

```

X.SHL.U.O:
  for i ← 0 to 128-size by size
    if  $c_{i+size-1..i+size-shift} \neq 0^{shift}$  then
      raise FixedPointArithmetic
    endif
     $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0^{shift}$ 
  endfor
X.SHR:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow c_{i+size-1}^{shift} \parallel c_{i+size-1..i+shift}$ 
  endfor
X.SHR.U:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow 0^{shift} \parallel c_{i+size-1..i+shift}$ 
  endfor
endcase
RegWrite(rd, 128, a)
enddef

```

FIG. 32C -3



Compress 32 bits to 16, with 4-bit right shift

FIG. 32D

Format

X.EXTRACT ra=rd,rc,rb

ra=xextract(rd,rc,rb)

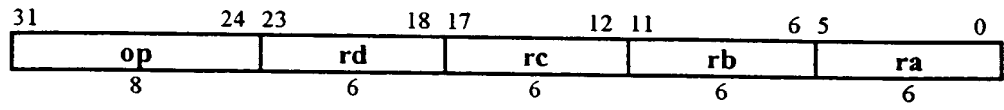


FIG. 33A

Definition

```

def CrossbarExtract(op,ra,rb,rc,rd) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case b8..0 of
    0..255:
      gsize ← 128
    256..383:
      gsize ← 64
    384..447:
      gsize ← 32
    448..479:
      gsize ← 16
    480..495:
      gsize ← 8
    496..503:
      gsize ← 4
    504..507:
      gsize ← 2
    508..511:
      gsize ← 1
  endcase
  m ← b12
  as ← signed ← b14
  h ← (2-m)*gsize
  spos ← (b8..0) and ((2-m)*gsize-1)
  dpos ← (0 || b23..16) and (gsize-1)
  sfsz ← (0 || b31..24) and (gsize-1)
  tfsz ← (sfsz = 0) or ((sfsz+dpos) > gsize) ? gsize-dpos : sfsz
  fsz ← (tfsz + spos > h) ? h - spos : tfsz
  for i ← 0 to 128-gsize by gsize
    case op of
      X.EXTRACT:
        if m then
          p ← dgsz+i-1..i
        else
          p ← (d || c)2*(gsz+i)-1..2*i
        endif
      endcase
    v ← (as & ph-1)||p
    w ← (as & vspos+fsz-1)gsz-fsz-dpos || vfsz-1+spos..spos || 0dpos
    if m then
      asz-1+i..i ← cgsz-1+i..dpos+fsz+i || wdpos+fsz-1..dpos || cdpos-1+1..i
    else
      asz-1+i..i ← w
    endif
  endfor
  RegWrite(ra, 128, a)
enddef

```

FIG. 33B

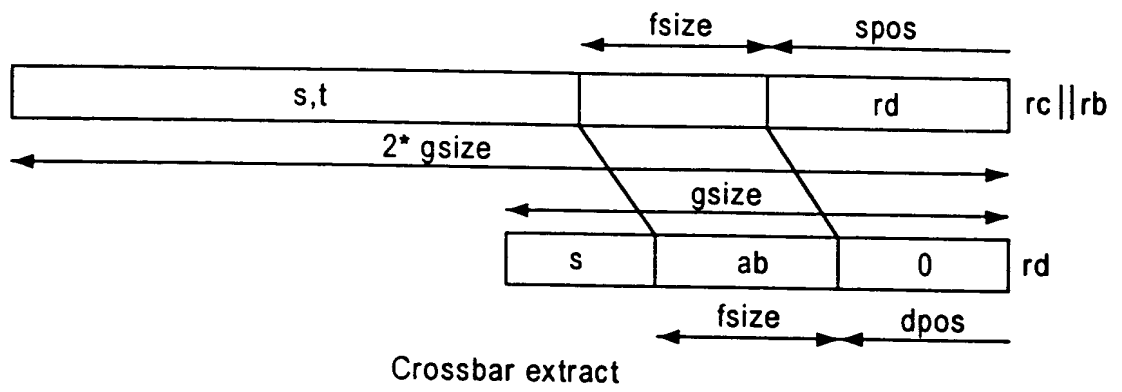


FIG. 33C

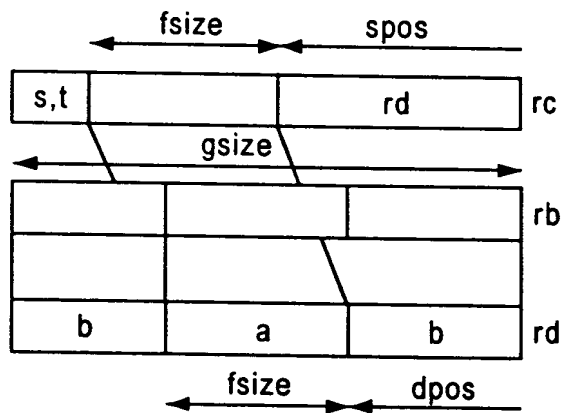


FIG. 33D

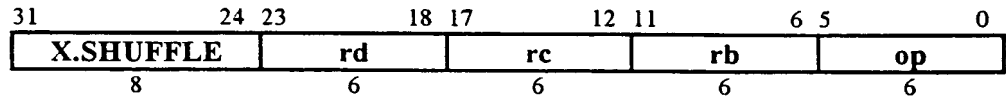
X.SHUFFLE.4	Crossbar shuffle within pecks
X.SHUFFLE.8	Crossbar shuffle within bytes
X.SHUFFLE.16	Crossbar shuffle within doublets
X.SHUFFLE.32	Crossbar shuffle within quadlets
X.SHUFFLE.64	Crossbar shuffle within octlets
X.SHUFFLE.128	Crossbar shuffle within hexlet
X.SHUFFLE.256	Crossbar shuffle within triclet

FIG. 34A

Format

X.SHUFFLE.256 rd=rc,rb,v,w,h
X.SHUFFLE.size rd=rcb,v,w

rd=xshuffle256(rc,rb,v,w,h)
rd=xshufflesize(rcb,v,w)



rc ← rb ← rcb
x←log₂(size)
y←log₂(v)
z←log₂(w)
op ← ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) + (size=256)*(h*32-56)

FIG. 34B

Definition

```

def CrossbarShuffle(major,rd,rc,rb,op)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  if rc=rb then
    case op of
      0..55:
        for x ← 2 to 7; for y ← 0 to x-2; for z ← 1 to x-y-1
          if op = ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) then
            for i ← 0 to 127
              ai ← c(i6..x || iy+z-1..y || ix-1..y+z || iy-1..0)
            end
          endif
        endfor; endfor; endfor
      56..63:
        raise ReservedInstruction
    endcase
  elseif
    case op4..0 of
      0..27:
        cb ← c || b
        x ← 8
        h ← op5
        for y ← 0 to x-2; for z ← 1 to x-y-1
          if op4..0 = ((17*z-z*z)/2-8+y) then
            for i ← h*128 to 127+h*128
              ai-h*128 ← cb(iy+z-1..y || ix-1..y+z || iy-1..0)
            end
          endif
        endfor; endfor
      28..31:
        raise ReservedInstruction
    endcase
  endif
  RegWrite(rd, 128, a)
enddef

```

FIG. 34C

Figure 35A

Wide Solve Galois

wminor	*galpoly	*galpoly	solv_par	wsolv_g
8	6	6	6	6

Solves $L*S = W \bmod z^{*8}$ in 8 iterations

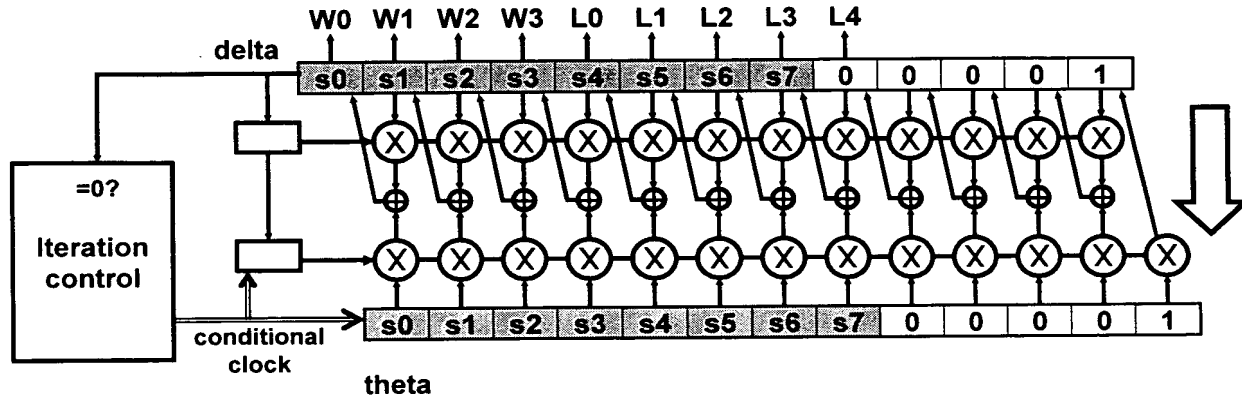


Figure 35B

Wide Solve Galois

```
static    v8_t    wsolveg(v8_t hh, v8_t syndrome, v8_t *omega)
```

```
for ( r=0; r < N_PARITY; r++)
```

```
{
    delta = _xcopyi8(delta0,0);
    delta0s = _castv8(_xshrm128(_castv128(delta0),_castv128(delta1),8));
    delta1s = _reindex8(delta1, -1);
    delta0 = _gxor8(_emulg8(gamma, delta0s, hh),_emulg8(delta,theta0, hh));
    delta1 = _gxor8(_emulg8(gamma, delta1s, hh),_emulg8(delta,theta1, hh));
    s = _gsetandne8(delta, _gsetge8(k,_gzero8));
    theta0 = _gmux8(s,delta0s,theta0);
    theta1 = _gmux8(s,delta1s,theta1);
    gamma = _gmux8(s,delta,gamma);
    k = _gmux8(s,_gnot8(k),_gadd8(k,_gone8));
}
lambda = _xselect8(delta1,delta0,USE_VCONST(lambdai));
*omega = _castv8(_xwithdrawu128(_castv128(delta0),64,0));
```

```
/*: A + 16*(B+A):*/
```

```
/*: 16*X :*/
```

```
/*: 16*X :*/
```

```
/*: 16*X :*/
```

```
/*: 16*(2*E+G) :*/
```

```
/*: 16*(2*E+G) :*/
```

```
/*: 16*2*G :*/
```

```
/*: 16*G :*/
```

```
/*: 16*G :*/
```

```
/*: 16*G :*/
```

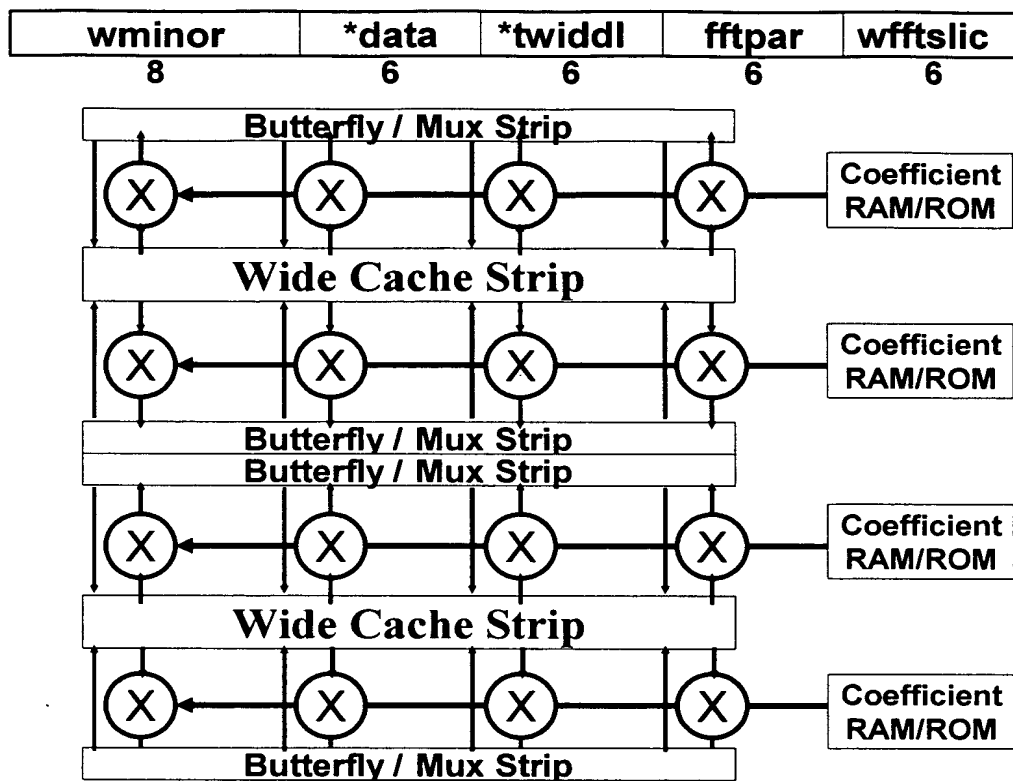
```
/*: 16*3*G :*/
```

```
/*: X :*/
```

```
/*: X :*/
```

Figure 36A

Wide FFT Slice



```

/*****
/* DSP library module : Inverse FFT, selectable length,          */
/*                      16-bit complex integers,                */
/*                      split-radix algorithm                    */
/*                                                                */
/*****

/* include files */
#include <stdio.h>
#include "broadmx.h"
#include "affirm.h"
#include "dspFFTud.h"
#include <math.h>

#define SHOW      0

/* typed version of _gboolean: should be part of gops */
static INLINE v16_t _gboolean16(v16_t src1, v16_t src2, v16_t src3, int imm)
{
    return _gboolean(src1.rr, src2.rr, src3.rr, imm).v16;
}

/* -----
 * I * (a - b) / 2
 */
static inline vc16_t _sub_mul_by_i_c16(vc16_t aa, vc16_t bb)
{
    v16_t muxmask      = _castv16(_gcopyi32(0xFFFF));
    v16_t xx;

    /* xx = _gsubh16n(_gmux16(muxmask,aa,bb),_gmux16(muxmask,bb,aa)); */
    xx = _gsubh16n(_gxor16(muxmask,bb),_gxor16(muxmask,aa));
    xx = _xswizzle16(xx, 7, 1);
    return xx;
}

```

Fig. 36B

```

/*-----
* Perform 4 independent 4-point fft's
*
* x0..x3 holds the input to the transform, 4 sets of 4 complex numbers.
* Each set is inverse-fourier transformed independently of the others.
* The results appear in x0..x3. The original values of y0..y3 are corrupted.
*/
#define QUAD_IFFT_4PT_c16(_y0,_y1,_y2,_y3, _x0,_x1,_x2,_x3) { \
    _y0 = _gaddh16n(_x0,_x2);          \
    _y1 = _gaddh16n(_x1,_x3);          \
    _y2 = _gsubh16n(_x0,_x2);          \
    _y3 = _sub_mul_by_i_c16(_x1,_x3);  \
    _x0 = _gaddh16n(_y0,_y1);          \
    _x2 = _gsubh16n(_y0,_y1);          \
    _x1 = _gaddh16n(_y2,_y3);          \
    _x3 = _gsubh16n(_y2,_y3);          \
}

/*-----
* Perform 4 independent 2-point fft's
*
* x0..x1 holds the input to the transform, 4 sets of 2 complex numbers.
* Each set is inverse-fourier transformed independently of the others.
* The results appear in y0..y1.
*/
#define QUAD_IFFT_2PT_c16(_y0,_y1, _x0,_x1) { \
    _y0 = _gaddh16n(_x0,_x1);          \
    _y1 = _gsubh16n(_x0,_x1);          \
}

```

Fig. 36B (cont)

```

static int _wfftlicecl6(vc16_t *dp, vc16_t *tp, int dn, int ds, int tn, int radix, int reorder, int extract)
{
    int i,j,ii, logmost;
    vc16_t *dwp, *twp;
    vc16_t t0,t1,t2,t3, d0,d1,d2,d3, p0,p1,p2,p3, z0,z1,z2,z3, m, n;

    if(SHOW) printf("extract = %d\n",extract&0xf);
    n = m = _gcopyil6(0);
    if (radix==4) {
        if (ds==1) {
            for (twp=tp,i=0; i<tn; dp++,twp++,i+=NELEMC16) {
                t0 = twp[0];
                d0 = dp[0];
                p0 = _emulx16(t0,d0,extract);
                z0 = _xshril6(p0,1);
                n = _gboolean16(n,p0,z0,0xf6);
                d0 = _vput16(d0,0,(_vget16(p0,0)+_vget16(p0,2)+_vget16(p0,4)+_vget16(p0,6)+2)>>2);
                d0 = _vput16(d0,1,(_vget16(p0,1)+_vget16(p0,3)+_vget16(p0,5)+_vget16(p0,7)+2)>>2);
                d0 = _vput16(d0,4,(_vget16(p0,0)-_vget16(p0,2)+_vget16(p0,4)-_vget16(p0,6)+2)>>2);
                d0 = _vput16(d0,5,(_vget16(p0,1)-_vget16(p0,3)+_vget16(p0,5)-_vget16(p0,7)+2)>>2);
                d0 = _vput16(d0,2,(_vget16(p0,0)-_vget16(p0,3)-_vget16(p0,4)+_vget16(p0,7)+2)>>2);
                d0 = _vput16(d0,3,(_vget16(p0,1)+_vget16(p0,2)-_vget16(p0,5)-_vget16(p0,6)+2)>>2);
                d0 = _vput16(d0,6,(_vget16(p0,0)+_vget16(p0,3)-_vget16(p0,4)-_vget16(p0,7)+2)>>2);
                d0 = _vput16(d0,7,(_vget16(p0,1)-_vget16(p0,2)-_vget16(p0,5)+_vget16(p0,6)+2)>>2);
                z0 = _xshril6(d0,1);
                m = _gboolean16(m,d0,z0,0xf6);
                dp[0] = d0;
            }
        } else {
            ii = ds / NELEMC16;
            for (twp=tp,i=0; i<tn; dp++,twp++,i+=4*NELEMC16) {
                t0 = twp[0*ii];
                t1 = twp[1*ii];
                t2 = twp[2*ii];
                t3 = twp[3*ii];
                for (dwp=dp,j=0; j<dn; dwp+=4*ii,j+=4*ds) {
                    d0 = dwp[0*ii];
                    d1 = dwp[1*ii];
                    d2 = dwp[2*ii];
                    d3 = dwp[3*ii];
                    d0 = _emulx16(t0,d0,extract); // can be eextract
                    d1 = _emulx16(t1,d1,extract);
                    d2 = _emulx16(t2,d2,extract);
                    d3 = _emulx16(t3,d3,extract);
                    z0 = _xshril6(d0,1);
                    z1 = _xshril6(d1,1);
                    z2 = _xshril6(d2,1);
                    z3 = _xshril6(d3,1);
                    n = _gboolean16(n,d0,z0,0xf6);
                    n = _gboolean16(n,d1,z1,0xf6);
                    n = _gboolean16(n,d2,z2,0xf6);
                    n = _gboolean16(n,d3,z3,0xf6);
                }
            }
        }
    }
}

```

Fig. 36B (cont)


```

    QUAD_IFFT_4PT_c16(p0,p1,p2,p3, d0,d1,d2,d3);
    z0 = _xshri16(d0,1);
    z1 = _xshri16(d1,1);
    z2 = _xshri16(d2,1);
    z3 = _xshri16(d3,1);
    m = _gboolean16(m,d0,z0,0xf6);
    m = _gboolean16(m,d1,z1,0xf6);
    m = _gboolean16(m,d2,z2,0xf6);
    m = _gboolean16(m,d3,z3,0xf6);
    dwp[0*ii] = d0;
    dwp[1*ii] = d1;
    dwp[2*ii] = d2;
    dwp[3*ii] = d3;
}
}
} else if (radix==2) {
    ii = ds / NELEMC16;
    for (twp=tp,i=0; i<tn; dp++,twp++,i+=2*NELEMC16) {
        t0 = twp[0*ii];
        t1 = twp[1*ii];
        for (dwp=dp,j=0; j<dn; dwp+=2*ii,j+=2*ds) {
            d0 = dwp[0*ii];
            d1 = dwp[1*ii];
            p0 = _emulx16(t0,d0,extract); // can be eextract
            p1 = _emulx16(t1,d1,extract);
            z0 = _xshri16(p0,1);
            z1 = _xshri16(p1,1);
            n = _gboolean16(n,p0,z0,0xf6);
            n = _gboolean16(n,p1,z1,0xf6);
            QUAD_IFFT_2PT_c16(d0,d1, p0,p1);
            z0 = _xshri16(d0,1);
            z1 = _xshri16(d1,1);
            m = _gboolean16(m,d0,z0,0xf6);
            m = _gboolean16(m,d1,z1,0xf6);
            dwp[0*ii] = d0;
            dwp[1*ii] = d1;
        }
    }
} else {
    for (j=0; j<dn; dp++,tp++,j+=NELEMC16) {
        *dp = d0 = *tp;
        z0 = _xshri16(d0,1);
        m = _gboolean16(m,d0,z0,0xf6);
    }
    n = m;
}
}

```

Fig. 36B (cont)

```

n = _gor16(n, _castv16(_xshriu128(_castv128(n), 64)));
n = _gor16(n, _castv16(_xshriu128(_castv128(n), 32)));
n = _gor16(n, _castv16(_xshriu128(_castv128(n), 16)));
logmost = _vget16(_elogmost16(n), 0);
if(SHOW) printf("logmost = %d (after mulx)\n", logmost);
m = _gor16(m, _castv16(_xshriu128(_castv128(m), 64)));
m = _gor16(m, _castv16(_xshriu128(_castv128(m), 32)));
m = _gor16(m, _castv16(_xshriu128(_castv128(m), 16)));
logmost = _vget16(_elogmost16(m), 0);
if(SHOW) printf("logmost = %d (after addh)\n", logmost);
return logmost;
}

static cplx16 const  exptab[][4] =
#define IFFT_COEFS_16
#include "dspIFFT-coefs.h"
#undef IFFT_COEFS_16
;

static void  make_twiddle(cplx16 *tw, int ni, int nj, int len, int show)
{
    int      ii, jj;

    for(ii = 0; ii < ni; ++ii) {
        for(jj = 0; jj < nj; ++jj) {
            tw->re = rint(-32768*cos(2*M_PI/len*ii*jj));
            tw->im = rint(-32768*sin(2*M_PI/len*ii*jj));
            if(show) printf("twiddle[%d][%d] = (%7d,%7d)\n", ii, jj, tw->re, tw->im);
            ++tw;
        }
    }
}

int dspInverseFourier_slice_c16(cplx16 *out, cplx16 const *in, int len)
{
    int logmost, extract, scale;
    static cplx16 twidtab[12][1024];
    int i, j, k, l;
    int ds, tn;

    for(i = 0; i < len; ++i) {
        twidtab[0][i].re = -32768;
        twidtab[0][i].im = 0;
    }
    make_twiddle(&twidtab[1][0], 4, 4, 16, 0);
    make_twiddle(&twidtab[2][0], 4, 16, 64, 0);
    make_twiddle(&twidtab[3][0], 4, 64, 256, 0);
    make_twiddle(&twidtab[4][0], 2, 256, 512, 0);

```

Fig. 36B (cont)

```

scale = 0;
logmost = 0;
if(len == 4) {
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)in, len, 0, 0, 1, 0, 0);
    scale = 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[0], len, 1, len, 4, 0, extract);
} else if(len == 16) {
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)in, len, 0, 0, 1, 0, 0);
    scale = 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[0], len, 1, len, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[1], len, 4, 16, 4, 0, extract);
} else if(len == 64) {
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)in, len, 0, 0, 1, 0, 0);
    scale = 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[0], len, 1, len, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[1], len, 4, 16, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[2], len, 16, 64, 4, 0, extract);
    scale -= 2;
} else if(len == 256) {
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)in, len, 0, 0, 1, 0, 0);
    scale = 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[0], len, 1, len, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[1], len, 4, 16, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[2], len, 16, 64, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslice16((vc16_t *)out, (vc16_t *)twidtab[3], len, 64, 256, 4, 0, extract);
    scale -= 4;
}

```

Fig. 36B (cont)

```

} else if(len == 512) {
    logmost = _wfftslicecl6((vc16_t *)out, (vc16_t *)in, len, 0, 0, 1, 0, 0);
    scale = 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslicecl6((vc16_t *)out, (vc16_t *)twidtab[0], len, 1, len, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslicecl6((vc16_t *)out, (vc16_t *)twidtab[1], len, 4, 16, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslicecl6((vc16_t *)out, (vc16_t *)twidtab[2], len, 16, 64, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslicecl6((vc16_t *)out, (vc16_t *)twidtab[3], len, 64, 256, 4, 0, extract);
    scale += 16 - logmost;
    extract = (1<<14) + (1<<13) + (2<<9) + (512-4*16+logmost+1);
    logmost = _wfftslicecl6((vc16_t *)out, (vc16_t *)twidtab[4], len, 256, 512, 2, 0, extract);
    scale -= 7;
}
if(SHOW) printf("scale = %d\n", scale);
return scale;

```

Fig. 36B (cont)

Format

W.CONVOLVE.X.order ra=rc,rd,rb

ra=wop(rc,rd,rb)

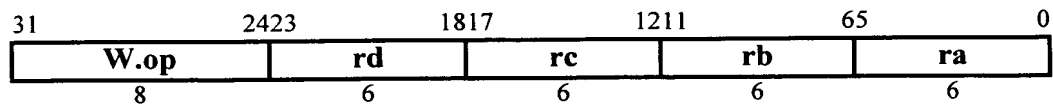


Fig. 37A

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
  mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsize-1+j)h-size || wsize-1+j..j)
enddef
```

```
def WideConvolveExtract(op,ra,rb,rc,rd)
  d ← RegRead(rd, 64)
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 128)
  case b8..0 of
    0..255:
      sgsz ← 128
    256..383:
      sgsz ← 64
    384..447:
      sgsz ← 32
    448..479:
      sgsz ← 16
    480..495:
      sgsz ← 8
    496..503:
      sgsz ← 4
    504..507:
      sgsz ← 2
    508..511:
      sgsz ← 1
  endcase
  l ← b11
  m ← b12
  n ← b13
  signed ← b14
  x ← b15
  if (c2..0 ≠ 0) or (d2..0 ≠ 0) then
    raise ReservedInstruction
  endif
  cwsz ← (c and (0-c)) || 05
  ct ← c and (c-1)
  cmsz ← (ct and (0-ct)) || 04
  ca ← ct and (ct-1)
  lcmsz ← log(cmsz)
  lcwsz ← log(cwsz)
  cm ← LoadMemory(c,ca,cmsz,order)
  dwsz ← (d and (0-d)) || 05
  dt ← d and (d-1)
  dmsz ← (dt and (0-dt)) || 04
  da ← dt and (dt-1)
  ldmsz ← log(dmsz)
  ldwsz ← log(dwsz)
  dm ← LoadMemory(d,da,dmsz,order)
  if (sgsz < 8) or (sgsz > wsize/2) then
    raise ReservedInstruction
```

```

endif
gsize ← sgsz
lgsize ← log(gsize)
case op of
    W.CONVOLVE.X.B:
        order ← B
    W.CONVOLVE.X.L:
        order ← L
endcase
cs ← signed
ds ← signed ^ m
zs ← signed or m or n
zsize ← gsize*(x+1)
h ← (2*gsize) + ldmsize - lgsize
spos ← (b8..0) and (2*gsize-1)
dpos ← (0 || b23..16) and (zsize-1)
r ← spos
sfsz ← (0 || b31..24) and (zsize-1)
tfsz ← (sfsz = 0) or ((sfsz+dpos) > zsize) ? zsize-dpos : sfsz
fsz ← (tfsz + spos > h+1) ? h+1 - spos : tfsz
if (b10..9 = Z) and not zs then
    md ← F
else
    md ← b10..9
endif
mzero ← b95..64
mpos ← b63..32
oo ← mpos || 03
ox ← oo|cwsz-1..lgsize
oy ← oo|cmsz-1..lcwsz
zz ← (~mzero) || 13
zx ← zz|dwsz-1..lgsize
zy ← zz|dmsz-1..ldwsz

```

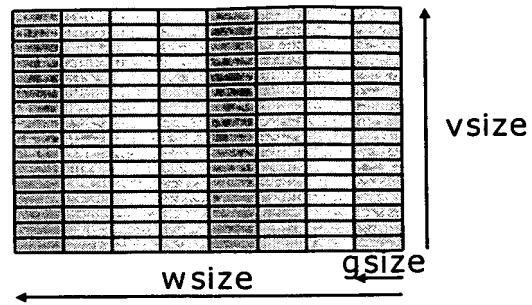
Fig. 37B (cont)

```

for k ← 0 to 128-zsize by zsize
  i ← k*gsize/zsize
  ix ← i|cwsz-1..lgsize
  iy ← i|cmsz-1..lcwsz
  q[0] ← 0h
  for j ← 0 to dmsize-gsize by gsize
    jj ← n and j|lgsize and not i|lgsize
    jx ← j|dwsz-1..lgsize
    jy ← j|dmsz-1..ldwsz
    u ← (oy+iy-jy)|cmsz-lcwsz-1..0 || (ox+ix-jx-2*jj)|cmsz-lcwsz-1..0 || 0lgsize
    if (jx>zx) or (jy>zy) and (dm|lgsize-1+j..j0) and undefined then
      q[j+gsize] ← q[j]
    else
      if jj then
        q[j+gsize] ← q[j] - mul(gsize,h,cs,cm,u,ds,dm,j)
      else
        q[j+gsize] ← q[j] + mul(gsize,h,cs,cm,u,ds,dm,j)
      endif
    endif
  endfor
  p ← q[dmsize]
  case rnd of
    none, N:
      s ← 0h-r || ~pr || ~prr-1
    Z:
      s ← 0h-r || ph-1
    F:
      s ← 0h
    C:
      s ← 0h-r || 1r
  endcase
  v ← ((zs & ph-1)||p) + (0||s)
  if (vh..r+fsize = (zs & vr+fsize-1)h+1-r-fsize) or not l then
    w ← (zs & vr+fsize-1)zsize-fsize-dpos || vfsize-1+r..r || 0dpos
  else
    w ← (zs ? (vhzsize-fsize-dpos+1 || ~vhfsize-1) : 0zsize-fsize-dpos || 1fsize) || 0dpos
  endif
  zzsize-1+k..k ← w
endfor
RegWrite(ra, 128, z)
enddef

```

Fig. 37B

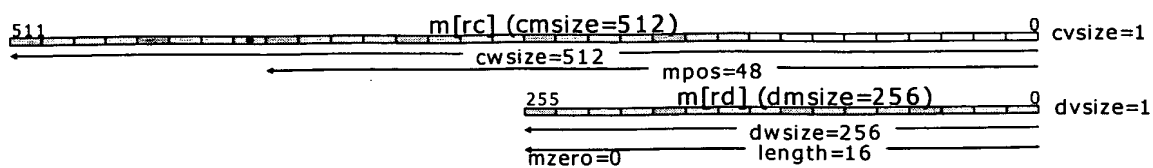


$$\text{msize} = \text{wsize} * \text{vsize}$$

$$\text{spec} = \text{base} + \text{msize}/16 + \text{wsize}/32$$

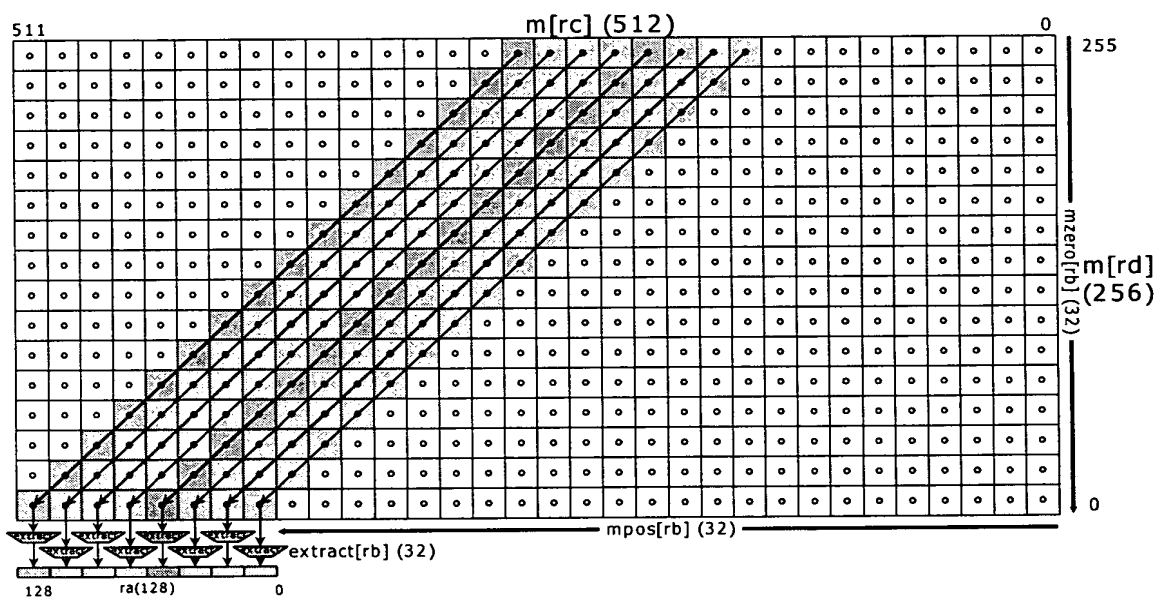
Wide operand specifier for wide convolve extract

Fig. 37C



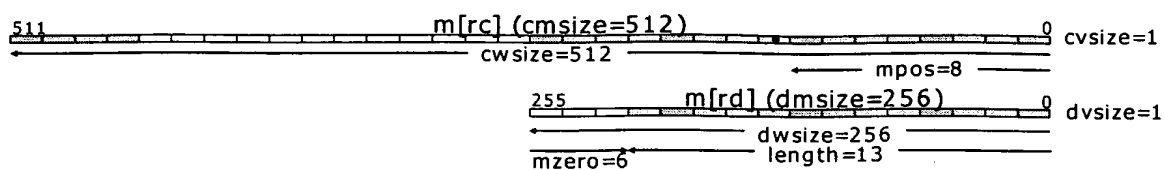
Wide convolve extract doublets

Fig. 37D



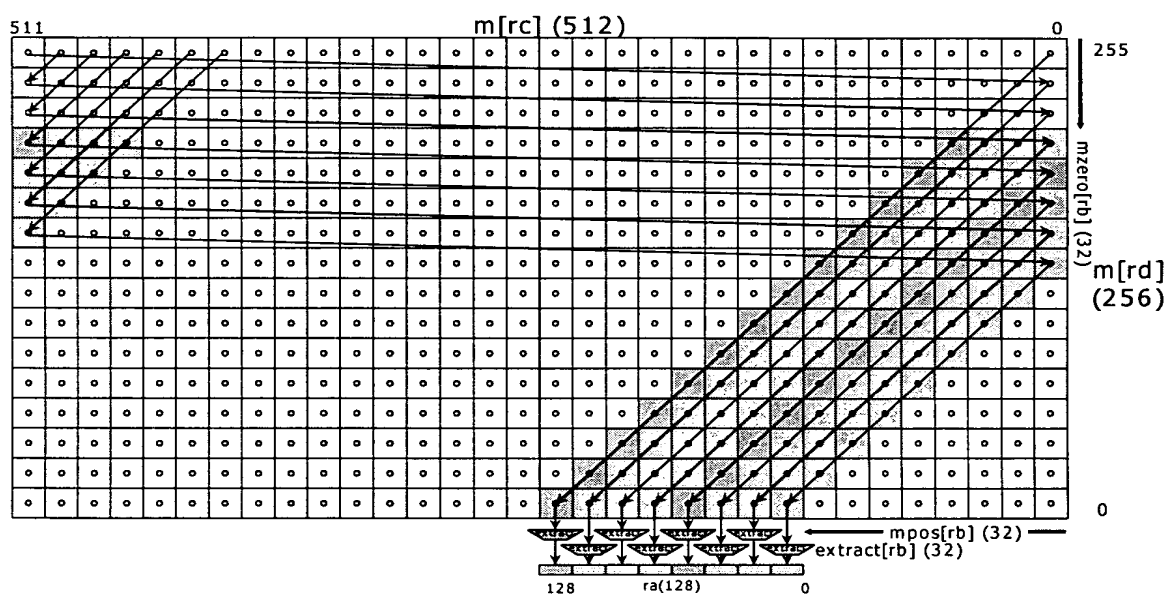
Wide convolve extract doublets

Fig. 37E



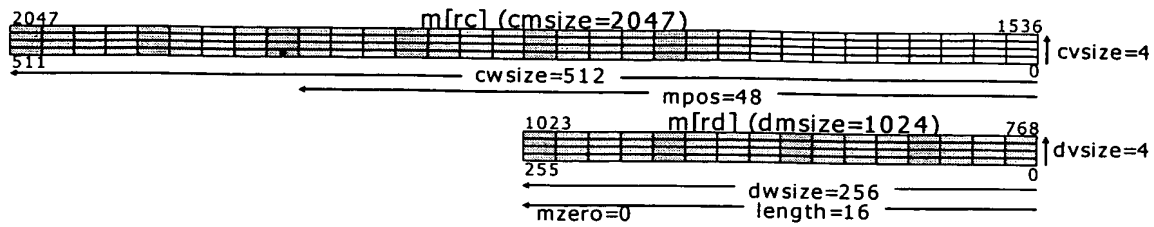
Wide convolve extract doublets

Fig. 37F



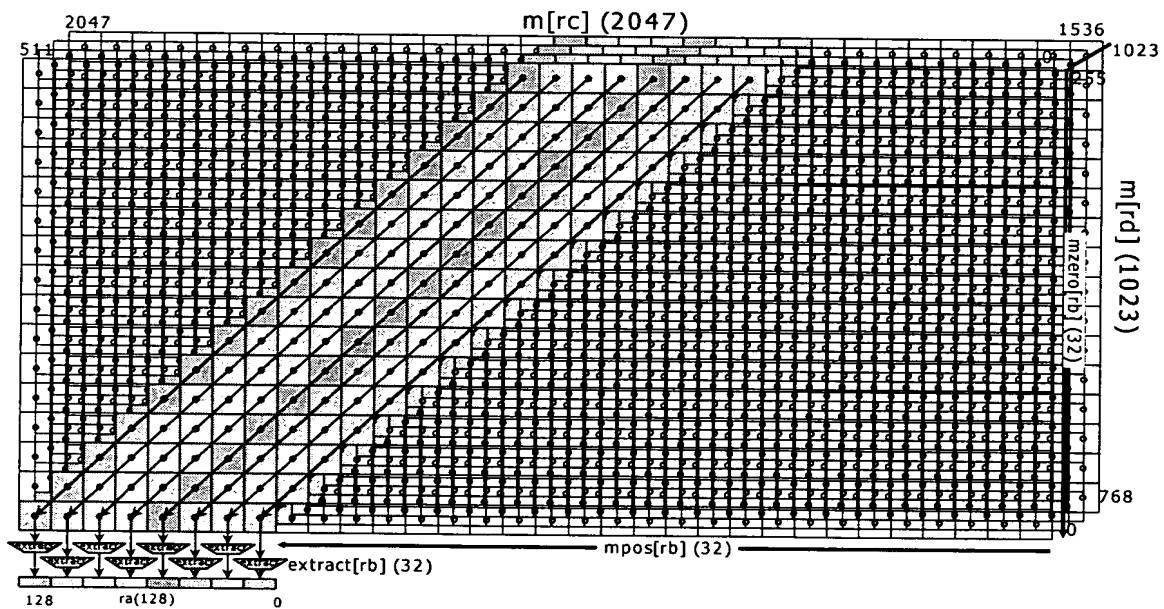
Wide convolve extract doublets

Fig. 37G



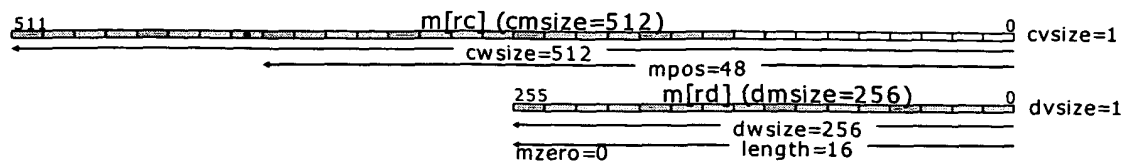
Wide convolve extract doublets two-dimensional

Fig. 37H



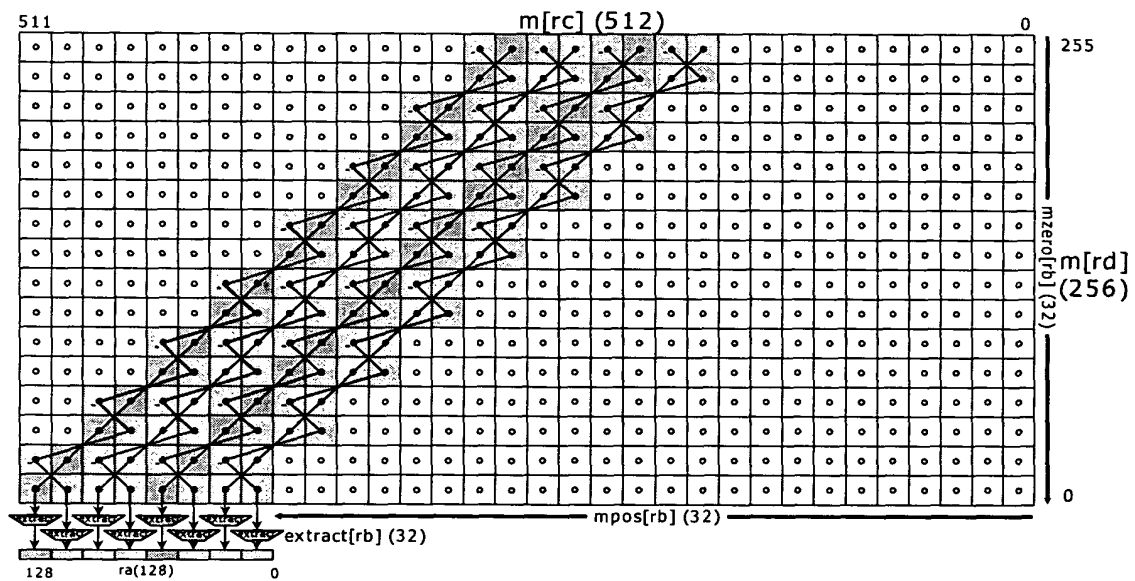
Wide convolve extract doublets two-dimensional

Fig. 37I



Wide convolve extract complex doublets

Fig. 37J



Wide convolve extract complex doublets

Fig. 37K

Figure 38 Wide Embedded Cache Coherency

